Masseuse*: Flexible Quality of Service for Internet Services

Abstract

In this paper we describe Masseuse*, a non-invasive approach to scalable quality-of-service provisioning that uses traffic shaping, admission control, and response monitoring at the border of an Internet site to ensure throughput and response time guarantees.

We experimentally compare an implementation of Masseuse both to hardware over-provisioning and to leading software approaches using real world workloads. Our results show that Masseuse can enforce the same QoS guarantees as either of the compared approaches, while achieving better resource utilization than over-provisioning and without the application rewriting overhead required by intrusive software approaches. We also demonstrate that our implementation can successfully handle extreme situations such as sudden traffic surges, application misbehavior and node failures. Furthermore, we demonstrate the flexibility of Masseuse by providing QoS guarantees for a complex and heterogeneous Internet service that cannot be implemented by other current software approaches.

1 Introduction

The current commercial importance of Internet services, make it imperative for companies relying on web-based technologies to offer and guarantee predictable, consistent, and differentiated quality of service (QoS) to their consumers. For example, e-commerce companies often want to provide faster response times for purchasing than for catalog browsing to ensure that no sale is lost due to the perception of an unresponsive transaction. Differentiated QoS also enables more general and flexible application hosting environments. For example, a service provider that hosts a personalized webmail portal for several companies wants to guarantee different levels of service to its customers and to ensure that these service guarantees are provided to each customer independently, regardless of overload or misbehavior of the others.

To meet large demand, scalable Internet services are commonly hosted using clustered architectures where a number of machines, rather than a single server, work together in a distributed and parallel manner to serve requests. Delivering reliable service quality guarantees in this distributed setting is the difficult challenge that our work addresses.

Both research and commercial Internet service communities have explored hardware-based and software-based approaches to QoS provisioning. The "state-of-the-practice" in current commercial settings is to deploy independent clusters for each service (hardware partitioning), each of which comprises enough capacity to service "worst-case" load conditions (over-provisioning). Unfortunately, because load fluctuations can be substantial, hardware partitioning and over-provisioning incurs a potentially high cost (sufficient resources must be available in each partition to handle load spikes) and low resource utilization (the extra resources are idle between spikes), making this approach inefficient.

As a result, software-based approaches have been proposed and developed to make better use of the resources employed to host Internet services. These approaches focus on embedding QoS logic at different levels of the site's internal software, including operating system [**?**, **?**, **?**, **?**], middleware [**?**, **?**, **?**], and application code [**?**, **?**, **?**]. It is the function of this logic to distribute, effectively, the workload among the cluster resources as a way of improving both resource utilization and client experience. Low-level techniques have been shown to provide a tight control on the utilization of resources (e.g., disk bandwidth or processor usage) while techniques that are closer to the application layer are able to satisfy QoS requirements that are more directly experienced by clients. However, these software solutions require the hosted application services and/or the hosting operating system to be customized for QoS provisioning, thereby limiting flexibility and extensibility. Furthermore, most current Internet sites include a myriad of different hardware and software platforms which are constantly evolving and changing. An invasive QoS solution that requires the reprogramming of hosted service code carries with it high development and testing costs when new services are introduced, or the existing site components (hardware and software) are reconfigured, upgraded, extended, etc. More problematically, the source code for many service components hosted at a site may not be available for proprietary reasons. This lack of source code makes the necessary software reprogramming remarkably difficult. Thus the growing complexity associated with Internet service hosting in commercial settings makes intrusive software QoS strategies less attractive as the need for extensibility and flexibility increases.

To address these needs, we propose a new approach to QoS provisioning for Internet services. Our approach offers reliable QoS guarantees at a lower cost than state-of-the-practice techniques, while giving the service providers the much needed flexibility that they require to rapidly reconfigure, upgrade and extend their complex set

*Our use of the name Masseuse is inspired by the observation that we are massaging both the input traffic to relieve the stress of the cluster and the title of the paper for the purposes of blind submission.

of services. In this paper we present *Masseuse*, a non-invasive software approach that treats the cluster and the services it is hosting as a "black-box" system and uses only feedback-driven techniques to control dynamically which and when each of the requests from the clients is forwarded into the cluster. Because traffic shaping and admission control is done at the entrance of the site, and the system uses only the observed request and response streams for its control algorithms, new services can be added, old ones upgraded, and resources reconfigured without re-engineering the necessary QoS mechanisms into the services themselves or the system software that supports them.

We report on an implementation of Masseuse and its experimental comparison with the state-of-the-practice (i.e., over-provisioning) and state-of-the-art (i.e., Neptune [**?**, **?**]) software solutions using realistic services, client request traces and clustered machines. Neptune [**?**, **?**] is a research and now commercially successful middleware system that implements QoS for Internet services, but which requires the services themselves to be rewritten to use Neptune primitives. Using the *Teoma* [**?**] search engine, which is explicitly programmed so it can use Neptune, we show that Masseuse can enforce the same QoS guarantees as Neptune for Neptune-enabled services, but without the additional engineering overhead associated with modifying the services that it supports. Furthermore, we illustrate Masseuse's ability to handle extreme situations such as sudden traffic surges, or internal application misbehavior – capabilities that are necessary for a successful deployment in large-scale, realistic settings. We also demonstrate the flexibility of Masseuse by showing how it can provide QoS guarantees for complex heterogeneous Internet services which cannot be modified – a capability that none of the published, pre-existing software approaches is capable of achieving at present.

1.1 Contributions

This paper makes five main contributions:

- We present Masseuse as a novel approach to QoS provisioning for large-scale Internet services that uses only observed input request and output response streams to control the load within the site so that quality guarantees are met.
- We describe a working implementation and demonstrate its viability using a large cluster system hosting commercial and community benchmark Internet services.
- We compare Masseuse with the best state-of-the-practice and state-of-the-art approaches in terms of efficiency and the degree to which they maintain QoS guarantees for both throughput and response times.
- We show the robustness of Masseuse in successfully overcoming extreme situations (i.e., sudden traffic surges, application misbehavior and node failures) which arise in current commercial settings.
- We demonstrate that the flexibility provided by Masseuse enables more efficient deployments of complex, heterogeneous Internet services than can currently be supported by existing approaches.

The remainder of this paper is organized as follows. Section 2 introduces Masseuse's approach and further describes its architecture. Section **??** experimentally compares Masseuse to the best of the known approaches. In Section **??** we demonstrate the robustness of Masseuse under extreme situations and also show its flexibility in providing reliable QoS guarantees in complex heterogeneous services. In Section 5 we discuss related work, and we conclude in Section **??**.

2 The Masseuse Architecture

To describe the Masseuse architecture, we begin by outlining the model of Internet service transactions we use. We treat Internet services (see Figure 1) as a stream of requests coming from clients that are received at the entrance of the site, processed by the internal resources, and returned back to the clients upon completion. In the case of system overload or internal error condition, requests can be dropped before completion and thus may not be returned to the client. Requests can be classified or grouped into different *service classes* according to a combination of service type and client identity. We view the QoS challenge as the ability to guarantee, at all times, a predefined quantitative characterization of the traffic in each service class as measured at the output of the cluster.

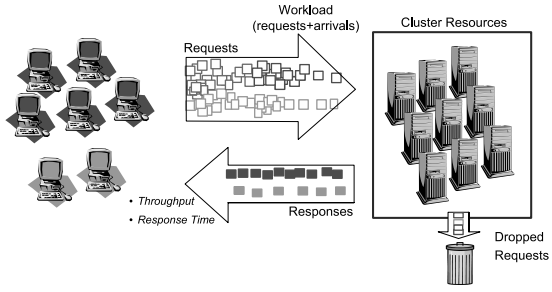

Figure 1: System model for Internet services.

In Masseuse, the QoS policy is specified as a list of *QoS classes* describing the quality that must be ensured for each class of service. We define QoS class as a tuple that

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95^{th}% ms)
BigCorp	`http://bigcorp.com/mail/*`	800	1000
SmallCorp	`http://smallcorp.com/mail/*`	50	200

Table 1: Example QoS policy for a service provider hosting webmail portals for two different companies.

describes: 1) how to identify requests of this class (*classification rules*) and, 2) what type of QoS must be enforced (*output guarantees*). In the same way as level-7 load-balancers [**?**, **?**, **?**], Masseuse classifies requests based on a combination of parameters such as IP address, port, URL and path. Output guarantees are specified in terms of guaranteed minimum throughput and maximum response time. For example, Table 1 describes a QoS policy containing two QoS classes for a service provider hosting webmail portals for two different companies. In the example, *BigCorp* has a much higher guaranteed throughput due to an expected higher traffic volume and *SmallCorp* requires much tighter response time guarantees for its users. Notice that the definition of output guarantees includes both throughput and response time requirements. While it is often possible to meet one type of guarantee at the expense of the other, our solution accommodates both. Additionally, Masseuse allows throughput and response time guarantees to be expressed using either percentiles or averages since the way in which each customer wishes to view a guarantee varies. In both cases, however, the time frame over which the average or percentile is computed is substantially longer than the time required to service an individual request.

Masseuse uses a single-policy enforcement engine to intercept and control in-bound traffic at the entrance of the site hosting the services. By tracking the responses to requests that are served within the site, our system automatically determines when new requests can be allowed entry such that a specified set of QoS guarantees will be enforced. No knowledge of the internals of the site are needed and no instrumentation is required. In other words, to make an Internet site capable of providing QoS guarantees it is enough to deploy Masseuse at its entrance point and define the desired QoS policy to be enforced.

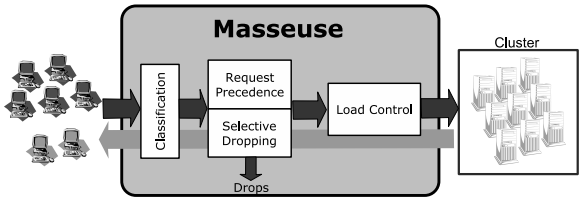

Figure 2: The architecture of Masseuse.

Figure 2 depicts the architecture of Masseuse, consisting of four different modules each of which implements part of the functionality that is necessary to enforce a QoS policy. The *Classification* module categorizes the intercepted requests from the clients into one of the service classes defined in the QoS class. The *Load Control* module determines the pace (for the entire system and all client request streams) at which Masseuse releases requests into the cluster. The *Request Precedence* module dictates the proportions with which requests of different classes are released to the cluster. The *Selective Dropping* module drops requests of a service class to avoid introducing work accumulation that would cause a QoS violation. This module also maintains responsiveness when the incoming service demands for a class exceed the processing capacity that it has been guaranteed. In the next sections we detail further the implementation of the Masseuse modules. We explicitly exclude the details associated with Classification since it is a well understood problem that has already been studied in the literature [**?**].

2.1 Load Control

The functionality of the Load Control module is twofold. First, it prevents large amounts of incoming traffic from overloading the internal resources of the cluster. When the internal resources become overloaded, the internal software (i.e., operating system, web servers, applications, etc.) will delay or drop requests without regard for their QoS classification. Second, it maintains the resources within the cluster at a high level of utilization to achieve good system performance. The goal of the Load Control module is to have the cluster operate at maximum capacity so that the largest possible capacity guarantees can be met, while also preventing overload conditions that would cause response time guarantees to be violated.

Our implementation exploits the direct correlation between the amount of work accumulation inside the cluster and the time required for requests to be computed by the hosted services. In general, more work introduced into the cluster corresponds to longer compute times for each service (given a fixed amount of resources) once the number of parallel requests exceeds the number of resources. With this in mind, the Load Control module can directly affect the amount of time that requests take to be computed inside the cluster (i.e., compute time) by controlling how much traffic is "in progress" at any time.

Similar to TCP, our implementation uses a sliding window scheme that defines the maximum number of requests that can be outstanding at any time (see Figure 4). The basic operation of the Masseuse engine consists of successively incrementing the size of the window until the compute times of the QoS class with the most restrictive response times approaches the limits defined by its guarantees . Our current implementation uses a simple algorithm (see Figure 3.a) that increments (or decrements) the window linearly until it observes a maximum compute time that is half the most restrictive of all the guarantees. The choice of 'half' is a compromise motivated by the

Load Control Algorithm

```
recalculateWindow()  // every Δ milliseconds
   CT(t) =  Observed Compute Times (t,t-Δ)
   RTG_min = Strictest Response Time Guarantee
   if CT(t) <  RTG_min/2 then
      window++  // linear increase
   else
      window--   // linear decrease
```

(a)

Request Precedence Scheduler

```
getNextRequest()    (minUsage = ∞)
   foreach class C
      //compute its weighted instant usage
      usage = C.outstanding/C.weight
      //save class with less usage
      if  usage<minUsage and ! C.isEmpty()
            targetClass = C ; minUsage = usage
   return targetClass.dequeueRequest()
```

(b)

Selective Dropping Algorithm

```
isDroppable(request)
   CT(t) =  Observed Compute Times (t,t-Δ)
   RT_max = Response Time Guarantee
   expectedRT = request.waitingTime + CT(t)
   if expectedRT > RT_max then
      return true  // Drop
   else
      return false // Forward
```

(c)

Figure 3: Simplified pseudo-code for the algorithms of the three main modules of Masseuse.

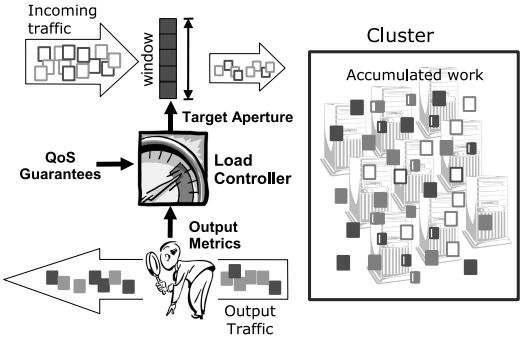

Figure 4: Structure of Load Control module.

tradeoff between the need to maintain cluster occupancy and limited internal queuing space necessary to absorb peaks of traffic. We are currently working on an optimized version that can dynamically adapt this threshold to allow more queuing without adversely affecting overall system performance. A more sophisticated (and reactive) version of the algorithm using non-linear variation of the window sizes is also under study.

2.2 Request Precedence

The main function of Request Precedence is to virtually partition the cluster resources among each of the service classes. Resource isolation is a necessary functionality that allows each service class to enjoy a minimum amount of processing capacity, independent of potential overload or misbehavior of others. This module is able to partition externally the service delivered by the cluster, by controlling the proportions in which the input traffic for each class is forwarded to the internal resources. Thus, the goal of this module is to ensure that the fraction of the overall cluster capacity devoted to each class is large enough to satisfy the guarantees for that class at all times.

The Request Precedence module also attempts to maximize performance in overload situations without allowing guarantees to lapse. It reassigns unclaimed resources to other QoS classes demanding more processing power than they have been granted. Reassigning unutilized capacity allows the QoS engine to take full advantage of the available cluster resources allowing some service classes to enjoy a level of service that is higher than what they have been guaranteed. At the same time, the Request Precedence module ensures that those classes that are not using their maximum allowable share of the overall capacity none-the-less receive enough capacity to meet their guarantees. By continually calculating and adjusting the fraction of cluster capacity that is given to each class, Masseuse differs from an approach that relies on physical partitioning of the resources where temporary reassignment cannot be implemented.

Under Masseuse, Request Precedence is implemented by a scheduling algorithm that logically partitions the window of outstanding requests (as dictated by the Load Control module) according to the throughput guarantees specified in the QoS classes. This method exploits (and depends on) the time-shared nature of current operating systems which assign time slices on resources equally amongst all running tasks. As a result, it is possible to increase the share of the cluster resources for a particular service class by increasing the number of tasks that are devoted to computing its requests. However, Masseuse has no direct knowledge of processes inside the cluster. Instead, by increasing or decreasing the number of outstanding requests for each class independently, the Request Precedence module can indirectly increase or decrease the proportions of resources allocated to each of them.

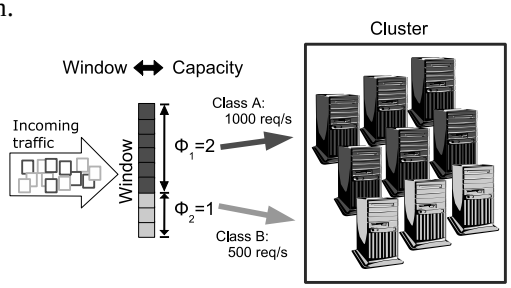

Figure 5: Function of Request Precedence module.

Our scheduler assigns a weight Φ_i to each of the QoS classes and uses this weight to partition proportionally the window accordingly (see Figure 5). Instead of allocating a fixed number of slots of the window per class, our algorithm (see Figure 3.b) uses a dynamic method that achieves similar characteristics to Weighted Fair Queuing

disciplines [**?**, **?**, **?**] in terms of proportional rate guarantees and reassignment of surplus. However the guarantees in our case apply to window sizes instead of service rates (i.e., throughput). The reason for this choice is that, throughput for a given service class can only be guaranteed when the computing requirements of the requests are known. In other words, the capacity necessary to achieve a given throughput is directly related to the computational complexity of the requests. On the other hand, assigning a particular window size corresponds to guaranteeing a portion of the cluster capacity, independently from the computing complexity of the incoming request stream. Therefore, in Masseuse the guarantees are computed in terms of capacity (i.e., resources) for each class, instead of minimum throughputs.

By working with a capacity measure (i.e., proportions of outstanding requests), Masseuse can provide effective isolation between classes when their computing requirements are not known *a priori* or can change dramatically. It uses capacity as a fungible metric that links output throughput and computing requirements such that an increase in one can be made to force a decrease in the other. For example a capacity equivalent to 10 nodes may correspond to an output throughput of 500 req/s at a compute cost of 20ms/req, but also to 1000 req/s if the compute cost is only 10ms/req. The internal capacity allocated for a class is calculated from the *nominal* guaranteed throughput (as expressed in the QoS class) and the *expected* computation requirements of the requests (as agreed upon between the provider and the consumer). In the cases where the computation complexity is violated (i.e., higher than agreed upon) for a particular class, instead of dropping the traffic of the faulty class, Masseuse will gracefully degrade its throughput to maintain the same internal capacity allocation.

2.3 Selective Dropping

The function of Selective Dropping is to discard the excessive traffic received for a QoS class in the situations where there is not enough available capacity to fulfill its incoming demands. A dropping module is necessary to prevent large delays from occurring in overloaded situations where requests would otherwise accumulate and then be dropped inside the cluster without regard to their QoS significance. That is, the Masseuse engine will control which requests are dropped so that all guarantees will be met. In the case where one class exceeds its allocated capacity, its requests should be targeted while requests for the "well-behaved" classes are allowed to proceed. As a result, the QoS guarantees will be observed for all requests of a class that are serviced, but if input load exceeds the maximum level that can be supported for the given guarantee, some requests will be dropped. Our Selective Dropping implementation ensures that the guarantees will be met for all requests that can be serviced. It does so by independently observing each of the QoS queues of the engine and discarding the requests that have been sitting in the queue for so long that the deadline for their service cannot be met. In our implementation (Figure 3.c), a request will be dropped if the time left for meeting the deadline once it gets at the head of the queue is less than the expected time of computation of its class. In other words, a request will be dropped if we expect it to miss its deadline according to how other requests of the same class are currently performing.

In Masseuse, Selective Dropping works closely with the Load Control module by signaling ahead of time when a service class is likely to become overloaded. This module leverages the queuing inside Masseuse to absorb safely peaks of traffic during transient overload conditions without violating the response time guarantees. For efficiency reasons, the module delays the dropping of requests to prevent discarding traffic in transient situations only to realize a moment later that the requests could have been served within the allowed response time limits. The implementation of independent dropping techniques, coupled with strong capacity guarantees given by the Request Precedence, allow this module to isolate response times of one class against misbehavior of others.

Combined, the functions of all four Masseuse modules (Classification, Load Control, Request Precedence and Selective Dropping) enable cluster responsiveness, efficient resource utilization, capacity isolation and delay differentiation, thus guaranteeing capacity and response times for each independent service class.

3 Experimental Performance Comparison

In this section we demonstrate that the four modules of Masseusecan provide QoS guarantees under realistic conditions even though they treat the cluster resources and Internet services as a "black-box". We have performed extensive studies of each of the presented modules, both in isolation as well as operating together. Due to space constraints we do not include them in this paper, but the details of these studies can be found in [**?**]. Instead, in this section we focus on examining the performance of Masseuse as a complete system, and study how it compares to the best of the known approaches. Our investigation is empirical and is based on the deployment of an Internet search service used by *Teoma* [**?**] using a 68-CPU cluster. We analyze how five different techniques (representing both state-of-the-practice and state-of-the-art) offer differentiated quality to distinct groups of customers using generated message traffic based on web-search traces. We then quantify the observed the quality of service delivered by each method.

3.1 Experimental Methodology

Our experimental setup consists of several client machines accessing a cluster system through an intermediate gateway/load-balancer machine. Accessing the services through a load balancer machine is the most commonly used architecture in current Internet services. For example, Google [**?**] funnels traffic through several Netscaler [**?**] load-balancing systems to balance the search load presented to each of its internal web servers [**?**].

To perform our experiments in the most realistic possible manner, we have deployed a commercial-grade Internet service on a 68-CPU cluster system and replayed real traffic traces from its commercial operation [**?**]. The service deployed is the index search component of the Teoma commercial search service [**?**]. The index search component consists of traversing an index database and retrieving the list of URLs that contain the set of words specified in the search query. The total size of the index database used is 12GB and is fully replicated at each node. The index search application from Teoma is specifically built for the Neptune middleware [**?**], a cluster-based software infrastructure that provides replication, aggregation and load balancing for network-based services. The version of Neptune we use also provides QoS mechanisms allowing the specification of proportional throughput guarantees and response times constraints through the definition of yield functions [**?**]. As it is the case with commercial search engines, our system accesses the service through a set of front-end machines that transform the received URLs into internal queries that are then forwarded to the middleware servicing the search database for processing. To mimic the environment at Teoma, we implement the front-end with an Apache web server [**?**] and a custom-built Apache module that interfaces with the Neptune infrastructure. This module is necessary to utilize the middleware functionality to locate other Neptune-enabled nodes and appropriately balance the requests based on the current load of the available servers. The cluster configuration used in our experiments is depicted in Figure 6. The hardware configuration of the cluster consists of 2.6 MHz Intel Xeon processors each with 3 gigabytes of main memory organized into nodes with either two or four processors per node. The network interconnect between processors is switched gigabit Ethernet and the host operating system is RedHat Linux/ Fedora Core release 1, using kernel version 2.4.24.

Our gateway node is a 4-CPU dedicated machine that can function in two different modes: as a load-balancer or as the Masseuse engine. When running in load-balancer mode, the machine is configured to implement the typical (Weighted) Round Robin and maximum connections options available in most commercial hardware [**?**, **?**, **?**].

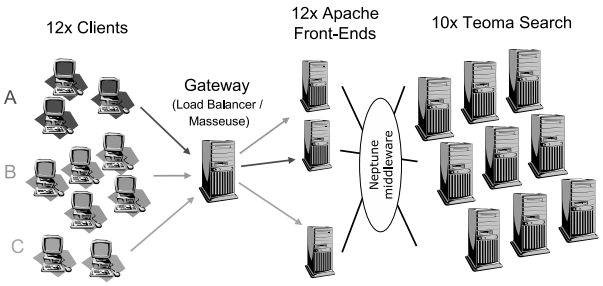

Figure 6: Experimental test-bed used for our benchmark using Teoma's search service.

When running as Masseuse engine, the gateway is configured to enforce the QoS policy defined for the experiment. Both the load-balancer and Masseuse engine are entirely implemented in user-level software. The gateway is implemented as an event-driven Java application which makes extensive use of the new libraries for improved I/O performance [**?**]. We use Sun's 1.5 Java virtual machine with low-latency garbage collection settings. Our performance tests show that our implementation can achieve a peak performance of 12Kreq/s (i.e., around 70K packets/sec) for certain client workloads. Thus the performance of our base-level system is high enough to be used in load levels that are comparable to current commercial systems (e.g. Google reports around 2500 req/sec [**?**], Ask Jeeves around 1000 req/sec [**?**]). Both our implementation of a load-balancer and the Masseuse engine are based on the same core software for fielding and forwarding HTTP requests.

For this experiment our methodology consists of using the previously described test-bed to recreate search traffic and to explore the effectiveness with which five different approaches can enforce a particular QoS policy for a single service with multiple client groups. The five compared approaches are:

Load Balancer The gateway machine is configured as a load balancer and tuned to match common high performance settings of Internet sites. Specifically, we configure it to use the least connections load-balancing algorithm and limit the maximum number of open connections for each front-end to match their configured maximum (i.e., 250 processes for Apache server and 150 for the Tomcat engine).

Physical Partitioning A separate group of machines are dedicated for each of the existing QoS classes. We configure the load-balancer to forward requests of a particular class only to its restricted set of reserved nodes.

Overprovisioning The size of each physical partition is increased such that the resulting capacity and response time guarantees can be achieved as specified

QoS Class	Classification	Output Guarantees		Experimental Workload	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (Avg. ms)	Input (Avg. req/s)	Service Status
A	Host: A	375	200	185	Not Overloaded
B	Host: B	937	600	1718	Overloaded
C	Host: C	562	300	557	Fully Utilized

Table 2: QoS guarantees and traffic workload of the Teoma search engine benchmark.

by the QoS policy (possibly at the expense of under utilized resources).

Neptune QoS The gateway is configured as a load balancer and the QoS mechanisms of Neptune are enabled to implement the QoS policy under study.

Masseuse QoS The gateway runs the Masseuse engine which implements QoS and the internal cluster resources implement only the Internet service. (i.e., QoS functionality in Neptune is disabled)

In order to benchmark Masseuse and the other considered QoS methodologies, client requests are replayed from a request trace supplied by Teoma that spans 3 different days of commercial operation [**?**]. We also use Teoma-supplied traces of word sequences to generate real search queries. The levels of incoming traffic are designed so that the input demands of the different clients are far below (class A), far above (class B) and coinciding with (class C) the capacity constraints specified in their respective QoS classes. Clients for each QoS class use different inter-arrival times, corresponding to one of the three different days of the original traces. Table 2 further depicts the details of the QoS policy and input workload used in the experiment, including the capacity and response time guarantees for each QoS class.

3.2 QoS Results

Figure 7 presents the results in terms of achieved average throughput and average response times for the five QoS methodologies using the same input request streams. The upper portion of the figure shows how the totality of incoming traffic for a class (represented by the height of a bar) has been divided into traffic that is served and traffic that is dropped. Horizontal marks delimit the minimum amount of traffic that has to be served if the QoS guarantees are met. Note that a resulting throughput below the horizontal marks still meets the QoS guarantee for a class if the totality of its incoming traffic is successfully served (i.e., the system cannot serve more traffic than it is received). The lower part of Figure 7 presents the results in terms of response times. For response times, we use horizontal marks to denote the maximum response times allowed by the QoS policy and denote with a darker color the classes that do not meet the guarantees. We present these response time results using a logarithmic scale for

Throughput (req/s)							
Class	Input	Guarantee	Load Balancer	Physical Partitioning	Overpro-visioning	Neptune QoS	Masseuse
A	185	375	148	185	185	185	185
B	1718	937	1333	986	1718	1133	1059
C	557	562	443	557	557	548	557

Response Times (ms)						
Class	Guarantee	Load Balancer	Physical Partitioning	Overpro-visioning	Neptune QoS	Masseuse
A	200	13335	40	40	45	121
B	600	13395	18937	75	256	600
C	300	13487	305	19	62	134

Table 3: Experimental results for Teoma search engine.

better visual comparison since the delays differ substantially. Table 3 summarizes these results in tabular form to further aid their comparison.

We begin by analyzing the quality of the service achieved by a load-balancer-only technique. Throughput results show that the amounts of traffic served in this case are directly dependent on the levels of incoming traffic rather than driven by the specified QoS policy, thus isolation between classes is not achieved. In this case we see that the dominance of class B traffic induces drops in A and C, even though the demands for these classes are always below (in the case of class A) or never exceed (for class C) the guaranteed capacity for each class. At the same time, the large response times shown in the lower figure, demonstrate that simple connection limiting techniques employed by the load-balancer are not enough to prevent large delays in response times (e.g. up to 14 seconds per request), rendering this technique inadequate to provide QoS guarantees.

When resources are physically dedicated through Physical Partitioning, the system is able to serve the expected amount of traffic for each of the classes and drop requests only in the cases when the demands of incoming traffic exceed the allocated capacity. Throughput guarantees are met, however, if we observe the results in terms of response time, we see that the overloaded partition B experiences a delay more that 30 times higher than the maximum allowed by the QoS policy. Thus while physically partitioning resources is able to provide capacity guarantees, it fails to ensure response times constraints for arbitrary incoming demands. It is worth noting that the reason for partition B serving more throughput than its guarantee is that the raw performance of the partition is slightly higher than the QoS guarantee defined in the policy.

When each of the partitions is augmented with enough resources (i.e., over-provisioning) all requests are successfully served. The response times are also reduced below the maximum allowed delay. In this case, class B and class C require an additional 10 and 2 CPUs respectively in order to meet the specified response time guarantees. Thus over-provisioning is the first of the techniques that can successfully provide both throughput and response time guarantees. However, meeting the QoS guarantees

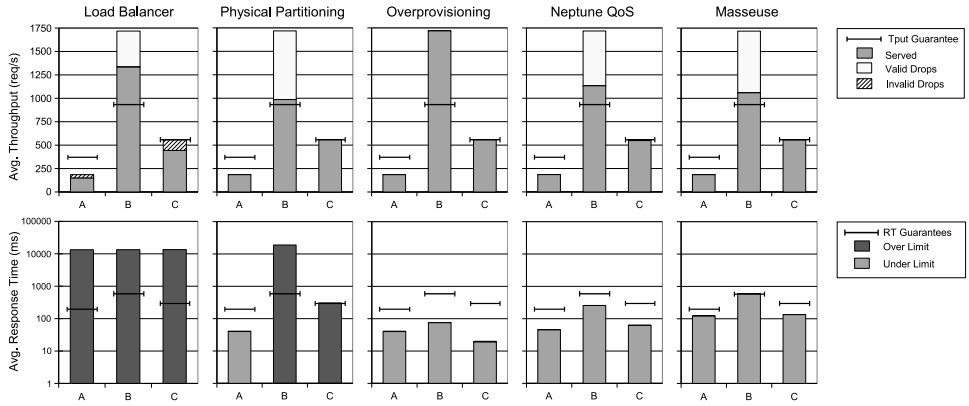

Figure 7: Experimental comparison of current approaches using Teoma's search engine.

through over-provisioning comes with a high cost. In our experiment, the increase in cost of overprovisioning was 60% (i.e., from 20 to 32 CPUs) with a resource utilization declining to 80%. Further, these numbers represent the *minimum* amount of over-provisioning that allowed us to achieve the QoS goals. In general, between load spikes the extra resources needed to serve surges in load lay idle. Thus, given the wide load fluctuations that most commercial Internet services experience we expect the resource utilization of over-provisioned systems *in situ* to become much worse than what we observe in this experiment.

Neptune QoS and Masseuse both meet the specified throughput and response time guarantees. Both techniques serve at least the necessary amount of traffic and are able to keep response time below the maximum delays associated with each guarantee. Furthermore, both techniques are able to successfully reassign the capacity not utilized by class A to the greedy clients of class B. We observe that direct control the resources and services in the cluster (due to its invasiveness) allows Neptune to achieve a slightly better throughput than Masseuse (i.e., 3%). This slight performance penalty can be seen as the cost that an external solution such as Masseuse has to pay for not modifying any of the software internals. However, given the completely non-invasive nature of Masseuse, we were surprised by how closely it matched the performance achieved by the invasive and commercially developed Neptune system. Figure 7 also shows that the resulting response times from Neptune are somewhat lower than Masseuse. This difference is because Masseuse is only designed to enforce maximum delay constraints and it is not concerned about minimizing the overall delay of service times. We are currently working on a prototype that can both ensure response time constraints and lower response delays when possible.

Summarizing, this experiment demonstrates the effectiveness of Masseuse empirically, using a commercial Internet service and commercial traffic levels. Masseuse in this setting is competitive with the best of the current approaches in its ability to enforce both response time and throughput QoS guarantees. In particular, Masseuse has less cost and achieves better resource utilization than over-provisioning techniques due to its ability to reassign unutilized capacity to those service classes that need it. At the same time, it achieves comparable QoS guarantees to an integrated and commercially available system such as Neptune, incurring only a small performance cost (i.e., 3%). In the next section (Section 4.4) we illustrate its flexibility by showing how it can provide reliable QoS guarantees in a complex and heterogeneous site running three different services.

4 Robustness under Extreme Conditions

In this section we investigate the robustness of Masseuse and its QoS enforcement capabilities under scenarios that emulate the extreme conditions experienced by many current Internet services. To do so, we first study the reaction of Masseuse to three circumstances: sudden traffic fluctuations (Section 4.1), sudden changes in computing requirements (Section 4.2) and node failures and recoveries (Section 4.3). We then present a larger-scale experiment in which we detail its response to the same conditions in a substantially more complex Internet hosting scenario (Section 4.4).

To conduct the initial set of isolated robustness studies we use two service classes: A and B. Service class A is a misbehaving class that begins with an input load that can be fully serviced with its allocated capacity, and then changes its demands to surpass the capacity required to meet its guarantees as well as to drive the overall system into overload. Service class B is a well-behaved class that receives a constant demand of traffic that is always below the traffic level that can be serviced under its guarantees. For each of the experiments, we detail how well Masseuse insulates the quality of service experienced by the well-behaved class B from the fluctuations introduced by class A. We also investigate how the quality of service given

to class A degrades gracefully during the periods when its demands exceed the capacity allocated to meet its guarantees. In particular, our goal is to provide as much capacity to A as possible without violating the guarantees made to either A or B. As described in subsection 2.2, however, the capacity allocated to A and B is fungible and constantly adjusted by Masseuse as it responds to changes in load conditions.

To run these experiments we use a system consisting of 4-CPUs for client machines accessing a 16-CPU cluster through a gateway machine implementing the Masseuse engine. Each of the servers runs the Tomcat application server [**?**], providing a "CPU-loop service" consisting of a servlet that loops a number of times so that it utilizes a certain amount of CPU (as specified in the HTTP parameters of each incoming request). This artificial emulation of a true web service allows precise control of the CPU load requirements associated with each request. Requests received from the clients are classified into QoS classes according to the host field name found in the HTTP header of the request (i.e., host: A or host: B).

The QoS policy defined for the experiments allocates the same guarantees for both classes of service (Table 4). Note that unlike the previous experiments, the response time guarantees are expressed in terms of 95th percentiles and not averages – a much more challenging but potentially more desirable metric to enforce, especially given the range of conditions to which we subject the cluster. All figures in this section depict the resulting average of the observed throughput (upper graph) and the 95th percentile of response times (lower graph) over two-second sampling intervals.

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95^{th}% ms)
A	Host: A	900	400
B	Host: B	900	400

Table 4: QoS policy used in the studies.

4.1 Sudden Traffic Fluctuations

In this experiment we show how Masseuse manages wide fluctuations of incoming traffic. To demonstrate this property we subject the service for class A to a sudden-but-sustained impulse of incoming traffic that is four times its normal rate. This sudden increase in demand is enough to bring the cluster to full utilization. Figure 8 shows the results from the experiment. In the Figure, the traffic fluctuation (labeled as "Input Class A") increases instantly from 600 req/s to 2400 req/s 120 seconds after the experiment has begun. Despite the sudden and sustained increase in A's traffic the degree to which service class B meets its guarantees is isolated from the change in input conditions. B's throughput is virtually unaffected and its

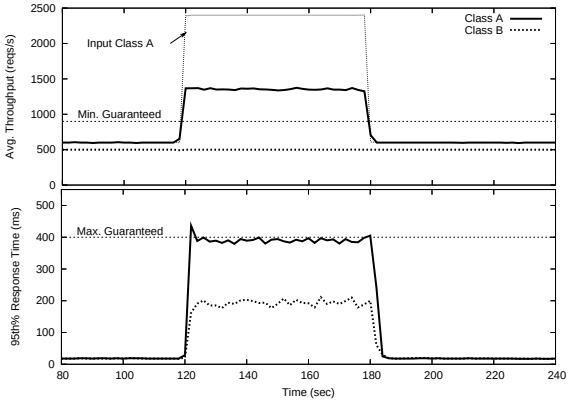

Figure 8: Masseuse's reaction to extreme fluctuation of incoming traffic.

response times, while they climb, are always kept below the maximum guaranteed delay. In response to the traffic surge, Masseuse quickly shifts any uncommitted resources to class A. Strictly speaking, it is consistent with the guarantee given to class A simply to cap throughput at 900 req/s for that class. However, by automatically sensing the degree to which is can slow down B's response times (without violating B's guarantees) and committing additional resources to A, Masseuse is able to give A as much throughput as can be spared while remaining within the constraints of both guarantees.

We should note that the slight spike in response times occurring in second 120 appears a consequence of our short sampling period. We wish to depict circumstances that stress the capabilities of Masseuse and as such, we calculate the percentiles with a two-second periodicity. In practice, it is unlikely that a commercial system will need to ensure QoS guarantees on such a fine-grained time scale, especially when using percentiles to specify guaranteed performance levels.

4.2 Computing Requirements Overload

In this experiment we investigate how Masseuse handles wide variations in the computing requirements associated with a request stream. These types of variations can occur in situations such as application misbehavior (e.g., software bugs that cause excessive resources to be used in computing a request) or changes in the workload characteristics (e.g., requests incurring in unusually long and expensive database queries). We induce this anomaly by suddenly increasing the computing requirements for class A from 8ms to 40ms of exclusive CPU time. Again, the goal is to protect the performance of class B while degrading the throughput given to class A to a level that is both maximal and consistent with the guarantees for both classes. To better observe the expected service for class A we include the throughput guarantees normalized

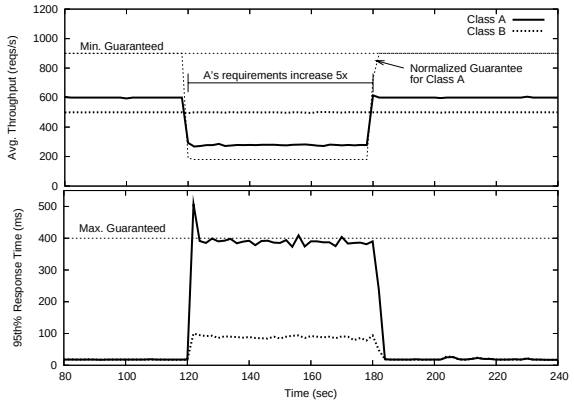

Figure 9: Behavior of Masseuse when requests of class A suddenly require five times more resources for their computation.

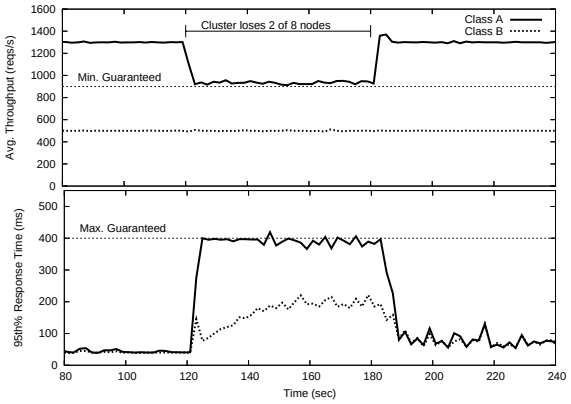

Figure 10: Masseuse's reaction to a failure of 2 nodes.

to its incoming computing requirements (i.e., the normalized throughput is five times lower than the nominal when requests are five times more difficult to compute).

Results from the experiment are depicted in Figure 9. As in the previous experiment the throughput given to class B remains virtually unaffected by the increase in computing requirements (seconds 120-180), and its response times are always kept below the guarantees. At the same time, in response to the increase in computing demands for the misbehaving class A, Masseuse immediately decreases A's throughput. Although degraded, A's throughput is always maintained above the normalized guarantee corresponding to the internal capacity allocation Masseuse made for this guarantee.

Recall from Section 2.2 the Request Precedence module guarantees enough resources to class A to fulfill the nominal throughput guarantee of 900 req/s assuming 8ms of computing time. When the computing requirements increase to 40ms/req the throughput must be lowered to 180 req/s to preserve enough capacity for B's guarantees. Thus we expect the system to enforce a throughput guarantee of 180 req/s for class A during the period in which its requests require 40ms of CPU time, as shown by the normalized guarantee line. However, between seconds 120 and 180 of the experimental period, class A is receiving a throughput of 280 req/s, which includes a surplus of 100 req/s corresponding to the resources that class B is not utilizing. If B's requirements were to suddenly increase, Masseuse would reduce A's throughput to 180 req/s and and change the proportion of B's requests admitted to reallocate more resources to B. Note also that this constant allocation and reallocation of capacity is sensed by the Masseuse engine automatically based on the observed responses leaving the cluster, and not based on predefined parameters or instrumentation describing the CPU requirements for each type of request. As is the case with the previous experiment, the short time scale over which each percentile is computed causes a single "spike" in response time during the two-second interval spanning second 120 in the trace.

4.3 Node Failures and Recoveries

In this experiment, we depict Masseuse's response to significant node failures and recoveries. At second 120, we induce the failure of 2 out of the 8 nodes and then recover the nodes 60 seconds later. To introduce these failures we program our load-balancer module to stop forwarding traffic to the "failed" nodes. We have also increased the incoming traffic rate for class A to 1300 req/s in order to make the resulting change in throughput more visible.

We show the results of the experiment in Figure 10. When the nodes fail, Masseuse rapidly reduces the throughput given to class A to its 900 req/s guarantee. Notice that this adjustment, again, does not violate the quality of the service guarantees given to class B. As with the previous two experiments, the throughput for B is unaffected while the response times grow to a level well below their maximum guaranteed delay.

We should note that in this example it was possible to enforce the QoS policy, even under the degraded operation, because there was enough spare capacity that B was not utilizing which could successfully be reassigned to A. In the cases where there are not enough resources to fulfill the guarantees across all classes, Masseuse reacts by degrading the service of each class proportionally to the guarantee associated with that class. For example, if the input demands for class B had been above the guaranteed 900 req/s, the Masseuse would have evenly assigned a throughput of 700 req/s for each class since the degraded capacity of the system would support 1400 req/sec in total, and the guarantees for both A and B are the same. We believe that other non-proportional mechanisms for reapportioning fungible capacity when QoS policies become infeasible are highly desirable and we plan to investigate them further in our future work.

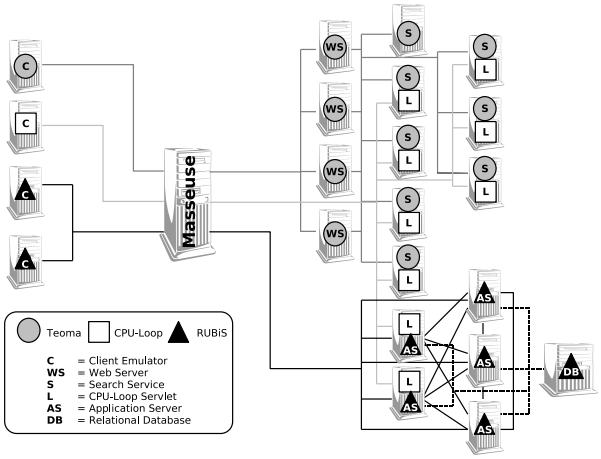

Figure 11: Setup of the complex, heterogeneous Internet site.

4.4 Complex Heterogenous Services

Through the previous set of controlled experiments we have shown that Masseuse can both enforce service isolation as well as gracefully degrade the service of misbehaving classes even under extreme operating conditions. We now show how Masseuse reacts to the same three severe circumstances for a larger-scale and substantially more complex Internet site that hosts three different services. Additionally, this experiment illustrates the flexibility of Masseuse's "black-box" approach: its ability to provide QoS guarantees using heterogeneous hardware configurations and multi-tiered software architectures where the source code of the applications cannot be modified. At present, we know of no other published infrastructure that can provide QoS for this complex Internet hosting scenario.

To perform this experiment we host the Teoma search and CPU-loop services (described previously) together with a third service called RUBiS [**?**] using shared set of cluster resources. RUBiS is a publicly available auction site modeled after eBay that has been used by several researchers for evaluating application server performance scalability [**?**, **?**]. We use the version of RUBiS that is implemented using Enterprise Java Beans (EJB) deployed on top of JOnAS application server (v3.3.6) and Tomcat (v4.1) servlet engine. The Tomcat servers are configured with session replication and the JOnAS application server is configured to balance the execution of EJBs across each of its nodes according to their respective loads. The auction data is stored using a mySQL database back-end with the same configuration and size as the benchmark described in [**?**]. Traffic for the RUBiS auction is generated by the client emulator supplied with the RUBiS software which performs typical user actions of an auction user such as browsing, bidding or buying items.

Figure 11 depicts the hardware and software configuration used for this experiment. Notice that we include both nodes that are dedicated to a single service as well as nodes that are shared by more than one service. In particular, the CPU-loop service shares 7 of the 8 nodes used by the Search component of Teoma, and also with 2 of the 5 nodes running the RUBiS auction. Our intention is to capture both the fluid sharing of cluster resources as well as the static capacity planning that we believe will always be present in a commercial system.

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95^{th}% ms)
Teoma	Port=8888	450	600
CPU-Loop	Port=9999	1350	3000
RUBiS	Port=10000	700	3500

Table 5: QoS policy for the complex and heterogeneous Internet site.

Also for this experiment we program our Masseuse engine with the QoS policy defined in Table 5, deploy it at the entrance of the site (with no other information than the QoS policy), and observe how well it performs in response to the same three types of changes explored in the previous subsections. Similarly, we generate three types of input load. For the Teoma service, we introduce incoming traffic that exceeds what can be completely serviced under the constraints of its guarantee. Alternatively, for the RUBiS service, we keep the incoming traffic load below the maximum serviceable level. We then vary the input for the CPU-loop service to create a peak of demand during the period from seconds 140 to 220 and to increase its computing requirements from 8ms to 40ms during the period between seconds 300 and 420. Finally we kill one of the Teoma back-end nodes at second 475 and restart it 120 seconds later.

Figure 12 shows the evolution of throughputs (above) and response times (below) for each of the three different services during the 11 minute run, in which a total of 1.1 million requests were served. Vertical lines separate the three different conditions (input increase, computation increase, node failure) to which Masseuse must respond. Throughput guarantees are again normalized to the expected computing requirements. Only CPU-loop service shows a deviation form the nominal throughput guarantees since it is the only service that suffers a change in its computation requirements. From the first segment of the figure, it is evident that Masseuse protects the RUBiS service and also reassigns the the available resources such that the two overloaded classes during the peak period are served according to the QoS policy. As we observed in Section 4.1, the amount of surplus service received by Teoma during the peak period, is given back to the CPU-loop service so that both classes can operate at their limits of throughput and response times.

In the second segment of the figure, the computing requirements of CPU-loop service increase to 5 times their original levels. In this case we induce a change in the computing requirements that it is more gradual than the

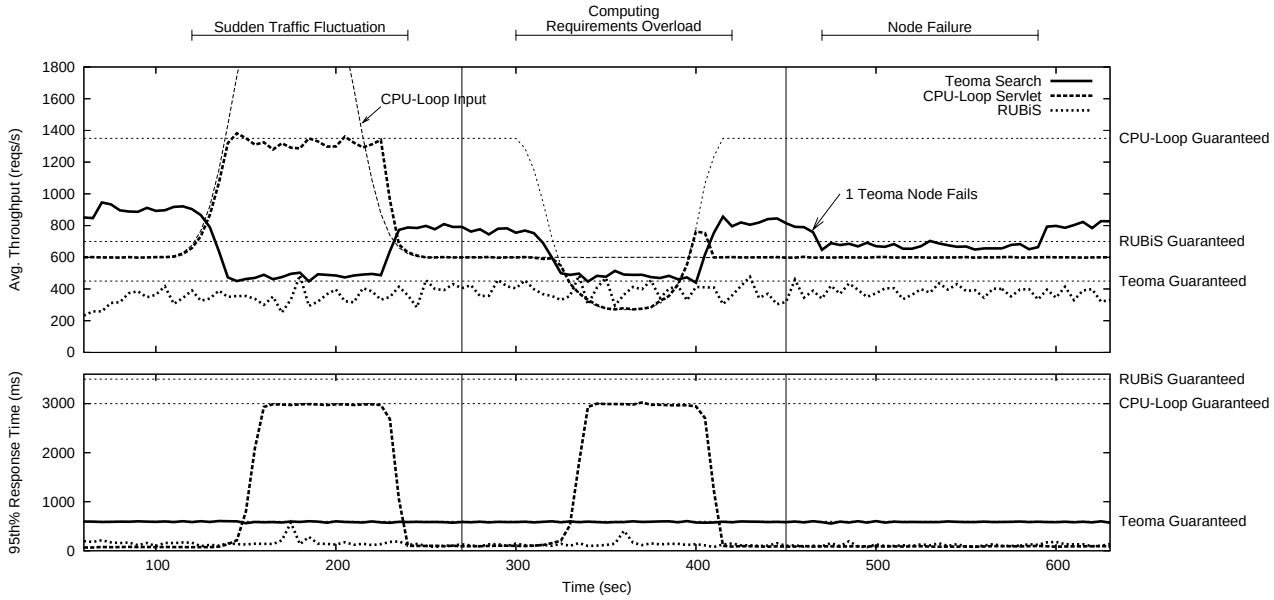

Figure 12: QoS results for a complex, heterogeneous Internet site.

sharp change shown in Section 4.2 to better emulate how a true Internet site might degrade. Masseuse reassigns capacity not needed to meet Teoma's guarantees to the CPU-loop service while maintaining the guarantees for RUBiS. Also, the CPU-loop service suffers a degradation in throughput that is inversely proportional to the increase in its computing requirements, thus maintaining the fungible capacity described by its guarantee. In this case, there are no extra resources to be used in aiding the overloaded CPU-loop class, thus its resulting throughput is capped exactly at its normalized guarantee.

In the third segment of the experiment the dedicated search back-end from the Teoma service fails. In this case we induce a *true failure* by killing the server process of Neptune and use the fail-over and recovery capabilities of the middleware to detect the change. Note that the failure of the node only has an effect in reducing the spare capacity that Teoma service is enjoying. Both the throughput and response times of CPU-loop and RUBiS are, once more, unaffected.

5 Related work

There are many approaches to providing QoS for Internet services, but relatively few that combine flexibility and extensibility with response time and throughput performance. In this section we briefly introduce some of the most relevant work and compare it to the Masseuse approach.

QoS for network communication is typically defined in terms of reliable communication between two endpoints with performance guarantees. Protocols such as diffserv [**?**] and intserv [**?**] leverage the existing routing infrastructure to provide bandwidth allocation and packet delay guarantees over the Internet. At a higher level, approaches such as Content Distribution Networks [**?**] provide similar features by appropriately managing an overlay network to content "closer" to the end-user. These approaches focus on the communication component and do not address the computational requirements associated with the servicing of Internet requests. In contrast Masseuse works at the boundary of the cluster hosting the services and, as such, complements approaches that ensure quality of network service between the client and the cluster.

Load balancers [**?**, **?**, **?**] are perhaps one of the the most closely related approaches to Masseuse. Properly tuned, load-balancers can greatly enhance the overall quality of the service offered by a cluster system. Products such as Packeteer [**?**] offer traffic shaping functionality such that minimum bandwidth guarantees can be allocated to distinct clients or applications. More sophisticated products such as Netscaler [**?**] apply intelligent connection management that protects the internal cluster nodes from overload in response to large bursts of incoming traffic. However, existing solutions are not aimed at providing throughput and response time guarantees, but are mainly designed to enhance the overall system performance. Futhermore, these techniques rely on the proper configuration of the load-balancers by an expert operator that knows and understands the internal operation of the site to be protected. As such, these are static configurations that are highly tuned for specific settings and that must be repeated for any change occuring in the site's internals. Masseuse differs from these approaches in that it guarantees QoS in terms of both throughput and re-

sponse times. At the same time Masseuse does not need to be configured explicitly or tuned by an expert for the specifics of the hardware or software of the site.

At the **operating systems** level, the QoS challenge is typically addressed in terms of resource management. Many research operating systems [**?**, **?**, **?**] achieve tight control on the utilization of resources as a way of enforcing capacity isolation between service classes. Although these techniques have proven to be effective in terms of capacity isolation, they are not designed to provide response time guarantees. Furthermore, these techniques control the resources within a single machine and thus cannot be easily extended to clustered environments. One notable exception is Cluster Reserves [**?**] – a single-node approach that has has been scaled to span clustered resources. Although this technique is shown to provide resource isolation at the cluster level, like its single-machine counterparts, it does not provide response time guarantees. Masseuse is also a cluster-wide QoS solution that provides both capacity and response time isolation as well as throughput and response time guarantees. It also differs from systems such as Cluster Reserves in that it does not require customization of the operating system used by the cluster's internal nodes.

Middleware systems such Neptune [**?**, **?**] or Application Server [**?**, **?**] include QoS functionality as part of a distributed and potentially scalable infrastructure. By programming the applications to use these primitives it is possible to construct distributed services that offer cluster-wide QoS guarantees. However in order for these frameworks to be effective each of the constituents of a service must be integrated with the middleware infrastructure. This often poses a very restrictive constraint given the heterogeneity and proliferation of current Internet services. Similar approaches that embed the QoS logic directly at the **Application level** have also been proposed. For example, the approach presented in SEDA [**?**] advocates the use of a specific framework for constructing well-conditioned scalable services and [**?**] shows the effectiveness of this framework when explicit QoS mechanisms are built to prevent overload in busy Internet servers. Rather than building an application with QoS support, other work has modified existing applications to include QoS capabilities [**?**, **?**]. For example, the work done in [**?**] shows how it is possible to modify the popular Apache web server to provide differentiated services without the use of resource management primitives at the operating system level. However, as is the case with middleware approaches, the large cost of modifying the application code to include QoS mechanisms is only effective if the entirety of the software deployment is able to function in a concerted way towards providing QoS. With Masseuse, the applications hosted in an Internet site do not need to be modified or designed for any particular operating system or middleware infrastructure and can directly be used in their native non-QoS state.

Some recent work has investigated resource management techniques using **Non-invasive approaches**. Façade [**?**] is a prototype implementation of a storage controller that throttles I/O requests to a (black-box) disk array. Similar to Masseuse, it provides response time isolation (but no throughput isolation) for different I/O streams. However, response time guarantees can only be enforced as long as the total incoming load is below the capacity of the disk array (i.e., no dropping mechanism is implemented). In [**?**], Jin et al. analyze the effectiveness of several share-based scheduling techniques for differentiating service quality in networked servers. Some of the project goals are similar in nature to Masseuse, however the analysis is done only through simulation, focuses only on storage server facilities and does not include a performance study in dynamic scenarios. Furthermore, the devised method is somewhat invasive since it requires offline profiling of the workload and more importantly assumes that the cost of every single requests can be known at scheduling time. Other work such as Gatekeeper [**?**] proposes a proxy system, much like Masseuse, that implements admission control for e-commerce applications. However, Gatekeeper is not designed provide any QoS guarantees, but targeted to reduce the overall response times and improve the performance of the system. Furthermore, it has only been tested in reduced size systems, it targets database back-ends and relies on extensive profiling of the service applications.

6 Conclusions and Future Work

Commercial Internet service provisioning depends increasingly on the ability to offer differentiated classes of service to groups of potentially competing clients. In addition, the services themselves may impose minimum QoS requirements for correct functionality. However, providing reliable QoS guarantees in large-scale Internet settings is a daunting task. Simple over-provisioning and physical partitioning of resources can be effective but inefficient. Invasive software approaches overcome the inefficiency problem but at the expense of reprogramming and/or re-engineering of the services within a site to implement QoS functionality.

In this paper we present an alternative, non-invasive software approach called Masseuse that provides efficient QoS provisioning for Internet services while allowing new levels of flexibility that current service providers require. The presented system functions at the border of an Internet site and uses traffic shaping, admission control, and response feedback to treat the site as a "black-box" control system. Masseuse intercepts the request and response streams entering and leaving a site to gauge how

and when new requests should be forwarded to the hosted services to ensure throughput and response time guarantees.

We demonstrate the capabilities of our Masseuse implementation by experimentally comparing it to the best state-of-the-practice and state-of-the-art approaches. Our results show that, despite being non-invasive, Masseuse can enforce the same QoS guarantees as either of the compared techniques, while achieving better resource utilization than over-provisioning and without the application rewriting overhead required by intrusive software approaches. We also demonstrate that our implementation can successfully handle extreme situations such as sudden traffic surges, application misbehavior or node failures. Further, we also demonstrate the powerful flexibility of Masseuse by providing QoS guarantees for a complex and heterogeneous Internet service that suffers the same type of harmful conditions. At present, we know of no other published infrastructure that can provide QoS under these challenging conditions. Encouraged by the performance of our results we are currently working on both enhancing the performance and scalability of the Masseuse engine as well as improving our algorithms with more sophisticated control mechanisms. Also we are interested in deploying Masseuse on a wider array of Internet services including real commercial sites.