Integrated Resource Management for Cluster-based Internet Services*

Kai Shen kshen@cs.ucsb.edu

Hong Tang htang@cs.ucsb.edu

Tao Yang tyang@cs.ucsb.edu

Lingkun Chu lkchu@cs.ucsb.edu

Dept. of Computer Science, University of California, Santa Barbara, CA 93106

Abstract

Client request rates for Internet services tend to be bursty and thus it is important to maintain efficient resource utilization under a wide range of load conditions. Network service clients typically seek services interactively and maintaining reasonable response time is often imperative for such services. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times. This paper presents an integrated resource management framework (part of *Neptune* system) that provides flexible service quality specification, efficient resource utilization, and service differentiation for cluster-based services. This framework introduces the metric of *quality-aware service yield* to combine the overall system efficiency and individual service response time in one flexible model. Resources are managed through a two-level request distribution and scheduling scheme. At the cluster level, a fully decentralized request distribution architecture is employed to achieve high scalability and availability. Inside each service node, an adaptive scheduling policy maintains efficient resource utilization under a wide range of load conditions. Our trace-driven evaluations demonstrate the performance, scalability, and service differentiation achieved by the proposed techniques.

1 Introduction

Previous studies show that the client request rates for Internet services tend to be bursty and fluctuate dramatically [5, 10, 11]. For example, the daily peak-to-average load ratio at Internet search service Ask Jeeves (`www.ask.com`) is typically 3:1 and it can be much higher and unpredictable in the presence of extraordinary events. As another example, the online site of Encyclopedia Britannica (`www.britannica.com`) was taken offline 24 hours after its initial launch in 1999 due to a site overload. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. As a consequence, it is important to maintain efficient resource utilization for those services under a wide range of load conditions.

Network clients typically seek services interactively and maintaining reasonable response time is imperative. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times, especially when the system is reaching its capacity limit and cannot provide interactive responses to all the requests. Quality of service (QoS) support and service differentiation have been studied extensively in network packet switching with respect to packet delay and connection bandwidth [12, 26, 37]. It is equally important to extend network-level QoS support to endpoint systems where service fulfillment and content generation take place. Those issues are especially critical for cluster-based Internet services in which contents are dynamically generated and aggregated [5, 17, 20, 33, 35].

This paper presents the design and implementation of an integrated resource management framework for cluster-based services. This framework is part of *Neptune* system: a cluster-based software infrastructure for aggregating and replicating partitionable network services [34, 35]. Neptune has been successfully deployed at Internet search engine Ask Jeeves [5] since December 2001. Although cluster-based network services have been widely deployed, we have seen limited research in the literature on comprehensive resource management with service differentiation support. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 2, 6, 7, 8, 27, 39] or clustered systems serving static HTTP content [3, 32]. In comparison, Neptune is intended for clustered services with dynamic service fulfillment or content generation. The work presented

*To appear in Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02).

in this paper addresses the inadequacy of the previous studies and complements them in the following three aspects.

Flexible resource management objectives. Most previous studies have been using a monolithic metric to measure resource utilization and define QoS constraints. Commonly used ones include system throughput, mean response time, mean stretch factor [41], or the tail distribution of the response time [28]. We introduce a unified quality-aware metric that links the overall system efficiency with individual service response time. To be more specific, we consider the fulfillment of a service request produces certain *quality-aware service yield* depending on the response time. The overall goal of the system is to maximize the aggregate service yield resulted from all requests. As an additional goal, the system supports service differentiation for multiple service classes.

Fully decentralized clustering architecture with quality-aware resource management. Scalability and availability are always overriding concerns for large-scale cluster-based services. Several prior studies have been relying on centralized components to manage resources for a cluster of replicated servers [3, 10, 32, 41]. In contrast, our framework employs a functionally symmetrical architecture that does not rely on any centralized components. Such a design not only eliminates potential single point of failure in the system, it is also crucial to ensuring smooth and prompt responses to demand spikes and server failures.

Efficient resource utilization under quality constraints. Neptune achieves efficient resource utilization through a two-level request distribution and scheduling scheme. At the cluster level, requests for each service class are evenly distributed to all replicated service nodes without explicit partitioning. Inside each service node, an adaptive scheduling policy adjusts to the runtime load condition and seeks high aggregate service yield at a wide range of load levels. When desired, the service scheduler also provides proportional resource allocation guarantee for specified service classes.

The rest of this paper is organized as follows. Section 2 illustrates a target architecture for this work and then describes our multi-fold resource management objective. Section 3 presents Neptune's two-level request distribution and scheduling architecture. Section 4 illustrates the service scheduling inside each service node. Section 5 presents the system implementation and trace-driven experimental evaluations. Section 6 discusses related work and Section 7 concludes the paper.

2 Targeted Architecture and Resource Management Objective

In this section, we first illustrate the targeted system architecture of this work. Then we introduce the concepts of quality-aware service yield and service yield functions. Through these concepts, service providers can express a variety of quality constraints based on the service response time. Furthermore, using service yield functions and resource allocation guarantees, our framework allows service providers to determine the desired level of service differentiation among multiple service classes.

2.1 Targeted Architecture

Neptune targets cluster-based network services accessible to many users through an intranet or the Internet. Inside those clusters, services are usually partitioned, replicated, aggregated, and then delivered to external clients through protocol gateways. *Partitioning* is introduced when the service processing requirement or data volume exceeds the capacity of a single server node. Service *replication* is commonly employed to improve the system availability and provide load sharing. Partial results may need to be *aggregated* across multiple data partitions or multiple service components before being delivered to external users.

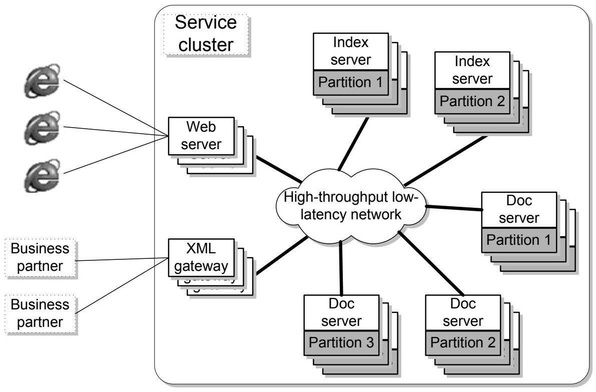

Figure 1: A targeted system architecture: search engine.

Figure 1 uses a prototype search engine to illustrate such a targeted system architecture [5, 18]. In this example, the service cluster delivers search services to consumers and business partners through Web servers and XML gateways. Inside the cluster, the main search tasks are performed on a set of index servers and document servers, both partitioned and replicated. Each search query first arrives at one of the protocol gateways. Then some index servers are contacted to retrieve the identifications of index documents related to the search query. Subsequently some document servers are mobilized to retrieve a short description of these documents and the final results are returned through the original protocol gateway. The resource management work in this study is focused on resources and quality constraints inside the service cluster. Issues related to wide-area network latency or bandwidth is out of the scope of this paper.

A large-scale service cluster typically consists of multiple groups of replicated service components. We call each replication group a *sub-cluster*. For instance, the replicas for partition 1 of the index servers in Figure 1 form one such sub-cluster. While Neptune supports the construction of multiple sub-clusters, this paper focuses on the resource management within a single sub-cluster. Here we give a brief discussion on the formation of sub-clusters. Each sub-cluster typically hosts a single type of service for modularity and ease of management. This scheme also allows for targeted resource allocation. For instance, machines with large number of CPUs can be allocated for sub-clusters hosting CPU-intensive service components while machines equipped with fast I/O channels can be used for sub-clusters hosting I/O-intensive components. Nonetheless, it is not uncommon to co-locate multiple types of service components in a single replication group to improve resource utilization efficiency.

2.2 Quality-aware Resource Utilization

Most previous studies have been using a monolithic metric such as system throughput, mean response time, mean stretch factor [41], or the tail distribution of the response time [28] to measure the efficiency of system resource management. We use a more comprehensive metric by conceiving that the fulfillment of a service request provides certain yield depending the response time. This yield, we call *quality-aware service yield*, can be linked to the amount of economical benefit or social reach resulted from serving this request in a timely fashion. Both goals of provisioning QoS and efficient resource utilization can be naturally combined as producing high aggregate yield. Furthermore, we consider the service yield resulted from serving each request to be a function of the service response time. The service yield function is normally determined by service providers to give them flexibility in expressing desired service qualities. Let $r_1, r_2, \cdots, r_k$ be the response times of the k service accesses completed in an operation period. Let $Y_i()$ represent the service yield function for the ith service access. The goal of our system is to maximize the aggregate yield, i.e.

$$\text{maximize} \sum_{i=1}^{k} Y_i(r_i). \tag{1}$$

In general, the service yield function can be any monotonically non-increasing function that returns non-negative numbers with non-negative inputs. We give a few examples to illustrate how service providers can use yield functions to express desired service qualities. For instance, the system with the yield function $Y_{\text{throughput}}$ depicted in Figure 2 (A) is intended to achieve high system throughput with a deadline D. In other words, the goal of such a system is to complete as many service accesses as possible with the response time $\leq D$. Similarly, the system with the yield function Y_{resptime} in Figure 2 (B) is designed to achieve low mean response time. Note that the traditional concept of mean response time does not count dropped requests. Y_{resptime} differs from that concept by considering dropped requests as if they are completed in D.

We notice that $Y_{\text{throughput}}$ does not care about the exact response time of each service access as long as it is completed within the deadline. In contrast, Y_{resptime} always reports higher yield for accesses completed faster. As a hybrid version of these two, Y_{hybrid} in Figure 2 (C) produces full yield when the response time is within a pre-deadline D', and the yield decreases linearly thereafter. The yield finally declines to a *drop penalty* C' when the response time reaches the deadline D. This corresponds to the real world scenario that users are generally comfortable as long as a service request is completed in D'. They get more or less annoyed when the service takes longer and they most likely abandon the service after waiting for D. C represents the full yield resulted from a prompt response and the drop penalty C' represents the loss when the service is not completed within the final deadline D. Figure 2 gives the illustration of these three functions. We want to point out that $Y_{\text{throughput}}$ is a special case of Y_{hybrid} when $D' = D$; and Y_{resptime} is also a special case of Y_{hybrid} when $D' = 0$ and $C' = 0$.

2.3 Service Differentiation

Service differentiation is another goal of our multi-fold resource management objective. Service differentiation is based on the concept of service classes. A *service class* is defined as a category of service accesses that obtain the same level of service support. On the other hand, service accesses belonging to different service classes

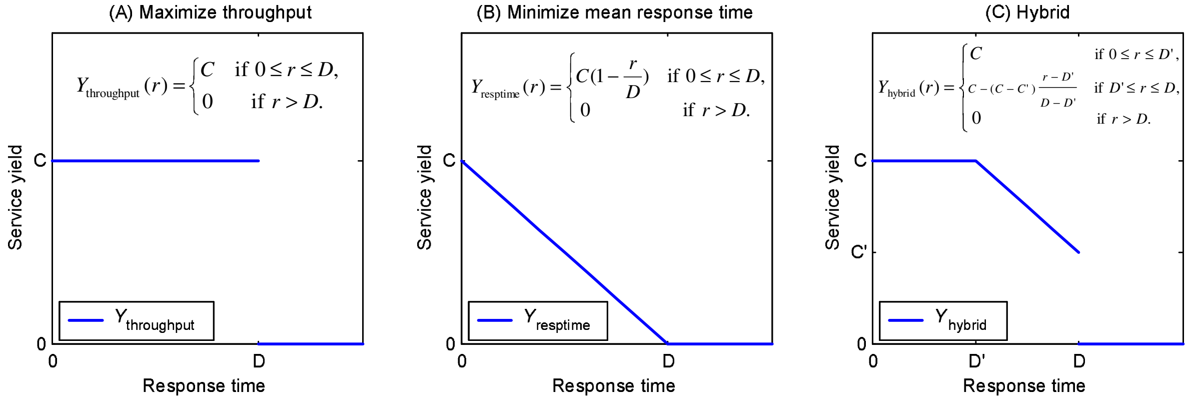

Figure 2: Illustration of service yield functions.

may receive differentiated QoS support. Service classes can be defined based on client identities. For instance, a special group of clients may be configured to receive preferential service support or a guaranteed share of system resources. Service classes can also be defined based on service types or data partitions. For example, a order placement transaction is typically considered more important than a catalog-browsing request.

We provide differentiated services to different service classes on two fronts. First, service classes can acquire differentiated service support by specifying different yield functions. For instance, serving a VIP-class client can be configured to produce higher service yield than serving a regular client. Secondly, each service class can be guaranteed to receive a certain portion of system resources. Most previous service differentiation studies have focused on one of the above two means of QoS support [7, 24, 30, 40]. We believe a combination of them provide two benefits when the system is overloaded: 1) the resource allocation is biased toward high-yield classes for efficient resource utilization; 2) a certain portion of system resources can be guaranteed for each service class, if needed. The second benefit is crucial to preventing starvation for low-priority service classes.

3 Two-level Request Distribution and Scheduling

In our framework, each external service request enters the service cluster through one of the gateways and it is classified into one of the service classes according to rules specified by service providers. Inside the cluster, service components are usually partitioned, replicated, and aggregated to fulfill the request. In this section, we discuss the cluster-level request distribution for a partition group or sub-cluster.

The dynamic partitioning approach proposed in a previous study adaptively partitions all replicas for each sub-cluster into several groups and each group is assigned to handle requests from one service class [41]. We believe such a scheme has a number of drawbacks. First, a cluster-wide scheduler is required to make server partitioning decisions, which is not only a single-point of failure, but also a potential performance bottleneck. Secondly, cluster-wide server groups cannot be repartitioned very frequently, which makes it difficult to respond promptly to changing resource demand. In order to address these problems, Neptune does not explicitly partition server groups. Instead, we employ a symmetrical and decentralized two-level request distribution and scheduling architecture illustrated in Figure 3.

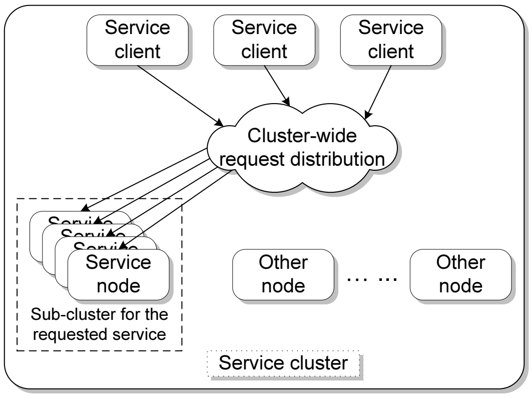

Figure 3: Two-level request distribution and scheduling.

In this scheme, each service node in a sub-cluster can process requests from all service classes. The resource management decision is essentially made at two levels. First, each service request is directed to one of the replicated service nodes through the cluster-level request distribution. Upon arriving at the service node, it is then subject to a node-level service scheduling. At the cluster level, Neptune employs a class-aware load balancing scheme to evenly distribute requests for each class to all replicas. Our load balancing scheme uses a random polling policy that discards slow-responding polls. Under this policy, whenever a client is about to seek a service for a particular service class, it polls a certain number of randomly selected service nodes to obtain the load information. Then it directs the service request to the node with the smallest number of active and queued requests. Polls not responded within a deadline are discarded. This strategy also helps excluding faulty nodes from request distribution. In practice, we use a poll size of 3 in our system. The polling deadline is set to be 10 ms, which is the smallest timeout granularity supported by `select` system call in Linux. Our recent study shows that such a policy is scalable and it performs well for services of small granularities [34]. Inside each service node, Neptune must also deal with the resource allocation across multiple service classes. This is handled by a node-level class-aware scheduling scheme, which will be discussed in Section 4.

An Alternative Approach for Comparison. For the purpose of comparison, we also design a request distribution scheme based on server partitioning [41]. Server partitioning is adjusted periodically at fixed intervals. This scheme uses the past resource usage to predict the future resource demand and makes different partitioning decisions during system under-load and overload situations.

- When the aggregate demand does not exceed the total system resources, every service class acquires their demanded resource allocation. The remaining resources will be allocated to all classes proportional to their demand.
- When the system is overloaded, in the first round we allocate to each class its resource demand or its resource allocation guarantee, whichever is smaller. Then the remaining resources are allocated to all classes under a priority order. The priority order is sorted by the full yield divided by the mean resource consumption for each class, which can be acquired through offline profiling.

Fractional server allocations are allowed in this scheme. All servers are partitioned into two pools, a dedicated pool and a shared pool. A service class with 2.4 server allocation, for instance, will get two servers from the dedicated pool and acquire 0.4 server allocation from the shared pool through sharing with other classes with fractional allocations.

The length of the adjustment interval should be chosen carefully so that it is not too small to avoid excessive repartitioning overhead and maintain system stability, nor is it too large to promptly respond to demand changes. We choose the interval to be 10 seconds in this paper. Within each allocation interval, service requests are randomly directed to one of the servers allocated to the corresponding service class according to the load balancing policy [34].

4 Node-level Service Scheduling

Neptune employs a multi-queue (one per service class) scheduler inside each node. Whenever a service request arrives, it enters the appropriate queue for the service class it belongs to. When resources become available, the scheduler picks a request for service. The scheduled request is not necessarily at the head of a queue. Figure 4 illustrates such a runtime environment of a service node.

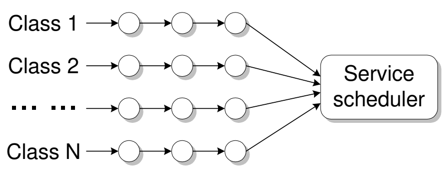

Figure 4: Runtime environment of a service node.

For a service node hosting N service classes: $C_1, C_2, \cdots, C_N$, each class C_k is configured with a service yield function Y_k and optionally a minimum system resource share guarantee g_k, which is expressed as a percentage of total system resources ($\sum_1^N g_k \leq 1$). The goal of the scheduling scheme is to provide the guaranteed system resources for all service classes and schedule the remaining resources to achieve high aggregate service yield. Figure 5 illustrates the framework of our service scheduling algorithm at each scheduling point. In the rest of this section, we will discuss two aspects of the scheduling algorithm: 1) maintaining resource allocation guarantee; and 2) achieving high aggregate service yield.

1. Drop from each queue head those requests that are likely to generate zero or very small yield according to the request arrival time, expected service time and the yield function.
2. Search for the service classes with non-empty request queues that have an estimated resource consumption of less than the guaranteed share. (Section 4.1)
   (a) If found, schedule the one with the largest gap between the resource consumption and the guaranteed share.
   (b) Otherwise, schedule a queued request that is likely to produce high aggregate service yield. (Section 4.2)

Figure 5: The node-level service scheduling algorithm.

4.1 Estimating Resource Consumption for Allocation Guarantee

In order to maintain resource allocation guarantee, we need to estimate resource consumption for each service class at each scheduling time. This estimation should be biased toward recent usage to stabilize quickly when the actual resource consumption jumps from one level to another. It should not be too shortsighted either in order to avoid oscillations or over-reactions to short-term spikes. Among many possible functions that exhibit those properties, we define the resource consumption for class C_k at time t to be the weighted summation of the resource usage for all class C_k requests completed no later than t. The weight is chosen to decrease exponentially with regard to the elapsed time since the request completion. For each request r, let $ct(r)$ be its completion time and $s(r)$ be its measured resource usage (we will discuss how to measure it in the end of this sub-section), which is known after its completion. Equation 2 defines $u_k(t)$ to be the resource consumption for class C_k at time t. Note that the time in all the following equations is denominated in *second*s.

$$u_k(t) = \sum_{\{r \mid r \in C_k \text{ and } ct(r) \leq t\}} \beta^{t-ct(r)} s(r), \qquad 0 < \beta < 1 \tag{2}$$

Another reason for which we choose this function is that it can be incrementally calculated without maintaining the entire service scheduling history. If we adjust $u_k(t)$ at the completion of every request and let t' be the previous calculation time, the resource consumption at time t can be calculated incrementally through Equation 3.

$$u_k(t) = \beta^{t-t'} u_k(t') + \beta^{t-ct(r)} s(r) \tag{3}$$

The selection of β should be careful to maintain the smooth and stable reaction for both short-term spikes and long-term consumption changes. In this paper we empirically choose β to be 0.95. Since we use *second* as the unit of time in those equations, this means a service request completed one second ago carries 95% the weight of a service request completed right now. With the definition of $u_k(t)$, the proportional resource consumption of class C_k can be represented by $\frac{u_k(t)}{\sum_{k=1}^{N} u_k(t)}$. In step 2 of the service scheduling, this proportional consumption is compared with the guaranteed share to search for under-allocated service classes.

This resource consumption estimation scheme is related to the exponentially-weighted moving average (EWMA) filter used as the round-trip time predictor in TCP [14]. It differs from the original EWMA filter in that the weight in our scheme decreases exponentially with regard to the elapsed time instead of the elapsed number of measurement samples. This is more appropriate for estimating resource consumption due to its time-decaying nature.

The detailed measurement of resource consumption $s(r)$ for each request r is application-dependent. Generally speaking, each request can involve mixed CPU and I/O activities and it is difficult to define a generic formula for all applications. Our approach is to let application developers decide how the resource consumption should be accounted. Large-scale service clusters are typically composed of multiple sub-clusters of replicated service components [5]. Each sub-cluster typically hosts a single type of service for modularity and ease of management. Thus requests in the same sub-cluster tend to share similar resource characteristics in terms of I/O and CPU demand and it is not hard in practice to identify a suitable way to account resource consumptions. In the current implementation, we use the accumulated CPU consumption for a thread or process acquired through Linux `/proc` file system. The effectiveness of this accounting model is demonstrated in our performance evaluation which contains a benchmark involving significant disk I/O.

Using single dimension resources simplifies our resource accounting model. We acknowledge that it could be desirable at times to co-locate CPU-intensive services with I/O-intensive applications to improve resource utilization efficiency. Accounting multi-dimension resources is not directly tackled in this paper. However, we believe a multi-dimensional resource accounting module can be added into our framework to address this issue.

4.2 Achieving High Aggregate Yield

In this section, we examine the policies employed in step 2b of the service scheduling to achieve high aggregated yield. In general, the optimization problem of maximizing the aggregate yield is difficult to solve given the fact that it relies on the advanced knowledge of the resource requirements of pending requests. Even for the offline case in which the cost for each request is known in advance, Karp has shown that the *Job Sequence* problem, which is a restricted case of our optimization problem, is NP-complete [25]. Various priority-based scheduling policies were proposed in real-time database systems to maximize aggregate realized value [22, 23]. Typical policies considered in those systems include Earliest Deadline First scheduling (*EDF*) and Yield or Value-Inflated Deadline scheduling (*YID*). EDF always schedules the queued request with the closest deadline. YID schedules the queued request with the smallest inflated deadline, defined as the relative deadline divided by the expected yield if the request is being scheduled.

Both EDF and YID are designed to avoid or minimize the amount of lost yield. They work well when the system resources are sized to handle transient heavy load [22]. For Internet services, however, the client request rates tend to be bursty and fluctuate dramatically from time to time [5, 10, 11]. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. During load spikes when systems are facing sustained arrival demand exceeding the available resources, missed deadlines become unavoidable and the resource management should instead focus on utilizing resources in the most efficient way. This leads us to design a *Greedy* scheduling policy that schedules the request with the lowest resource consumption per unit of expected yield. The Greedy method typically performs well when the system is overloaded. However, it is not optimal because it only maximizes the efficiency for the next scheduled request without considering longer impact of the scheduling decision.

In order to have a scheduling policy that works well at a wide range of load conditions, we further design an *Adaptive* policy that dynamically switches between YID and Greedy scheduling depending on the runtime load condition. The scheduler maintains a 30-second window of recent request dropping statistics. If more than 5% of incoming requests are dropped in the watched window, the system is considered as overload and the Greedy scheduling is employed. Otherwise, the YID scheduling is used.

All the above scheduling policies are priority-based scheduling with different definition of priorities. Table 1 summarizes the priority metrics of the four policies. Three of these policies require a predicted service time and resource consumption for each request at the scheduling time. For the service time, we use an exponentially-weighted moving average of the service time of past requests belonging to the same service class. Resource consumption measurement is application-dependent as we have explained in the previous sub-section. In our current implementation, such a prediction is based on an exponentially-weighted moving average of the CPU consumptions of past requests belonging to the same service class.

Policy	**Priority** (the smaller the higher)
EDF	Relative deadline
YID	Relative deadline divided by expected yield
Greedy	Expected resource consumption divided by expected yield
Adaptive	Dynamically switch between YID (in under-load) and Greedy (in overload)

Table 1: Summary of scheduling policies.

5 System Implementation and Experimental Evaluations

Neptune has been implemented on a Linux cluster. In addition to the resource management framework described in this paper, Neptune provides load balancing and service replication with fail-over support for cluster-based services [34, 35]. Application developers can easily deploy services through specifying a set of RPC-like access methods for each service and clients can access them through a simple programming API. Neptune employs a symmetrical architecture in constructing the service infrastructure. Any node can elect to provide services and seek services from other nodes inside the service cluster. Each external service request is assigned a service class ID upon arriving at any of the gateways. Those requests are directed to one of the replicated service nodes according to the class-aware load balancing scheme. Each server node maintains multiple request queues (one per service class) and a thread pool. To process each service request, a thread is dispatched to invoke the application service component through dynamically-linked libraries. The size of the thread pool is chosen to strike the balance between concurrency and efficiency depending on the application characteristics. The aggregate services are exported to external clients through protocol gateways. Neptune was subsequently ported into Solaris platform. An earlier version of Neptune has been successfully deployed at Internet search engine Ask Jeeves [5] since December 2001. The resource management framework described in this paper, however, has not yet been incorporated into the production system.

The overall objective of the experimental evaluation is to demonstrate the performance, scalability, and service differentiation achieved by the proposed techniques. In particular, the first goal is to examine the system performance of various service scheduling schemes over a wide range of load conditions. Secondly, we will study the performance and scalability of our cluster-level request distribution scheme. Our third goal is to investigate the system behavior in terms of service differentiation during demand spikes and server failures. All the evaluations were conducted on a rack-mounted Linux cluster with 30 dual 400 MHz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

5.1 Evaluation Workloads

Our evaluation studies are based on two service workloads. The first service is a *Differentiated Search* service based on an index search component from Ask Jeeves search. This service takes in a group of encoded query words; checks a memory mapped index database; and returns a list of URLs that are relevant to input query words. The index database size is around 2.5 GB at each node and it cannot completely fit in memory. The mean service time for this service is around 250 ms in our testbed when each request is served in a dedicated environment.

Differentiated Search distinguishes three classes of clients, representing Gold, Silver, and Bronze memberships. We let the request composition for these three classes be 10%, 30%, 60% respectively. The yield functions of these service classes can be one of the three forms that we described in Section 2.2, i.e. $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, or $Y_{\text{hybrid}}()$. In each case, the shapes of the yield functions for three service classes are the same other than the magnitude. We determine the ratio of such magnitudes to be 4:2:1. The deadline D is set to be 2 seconds. In the case of $Y_{\text{hybrid}}()$, the drop penalty C' is set to be half of the full yield and the pre-deadline D' is set to be half of the absolute deadline D. Figure 6 illustrates the yield functions when they are in each one of the three forms.

The request arrival intervals and the query words for the three Differentiated Search service classes are based on a one-week trace we collected at Ask Jeeves search via one of its edge Web servers. The request distribution among the edge Web servers are conducted by a balancing switch according to the "least connections" policy. Note that this trace only represents a fraction of the complete Ask Jeeves traffic during the trace collection period. Figure 7 shows the total and non-cached search rate of this trace. The search engine employs a query cache to directly serve those queries that have already been served before and cached. We are only concerned with non-cached requests in our evaluation because only those requests invoke the index search component. We use the peak-time portion of Tuesday, Wednesday, and Thursday's traces to drive the workload for Gold, Silver, and Bronze classes respectively. For each day, the peak-time portion we choose is the 7-hour period from 11am to 6pm EST. The statistics of these three traces are listed in Table 2. Note that the arrival intervals of these traces may be scaled when necessary to generate workloads at various demand levels during our evaluation.

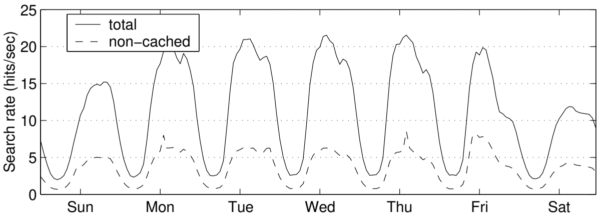

Figure 7: Search requests to Ask Jeeves search via one of its edge web servers (January 6-12, 2002).

	Number of accesses	Arrival interval	
	Total (non-cached)	Mean	Std-dev
Gold	507,202 (154,466)	161.3ms	164.3ms
Silver	512,227 (151,827)	166.0ms	169.5ms
Bronze	517,116 (156,214)	161.3ms	164.7ms

Table 2: Statistics of evaluation traces.

The three service classes in Differentiated Search are

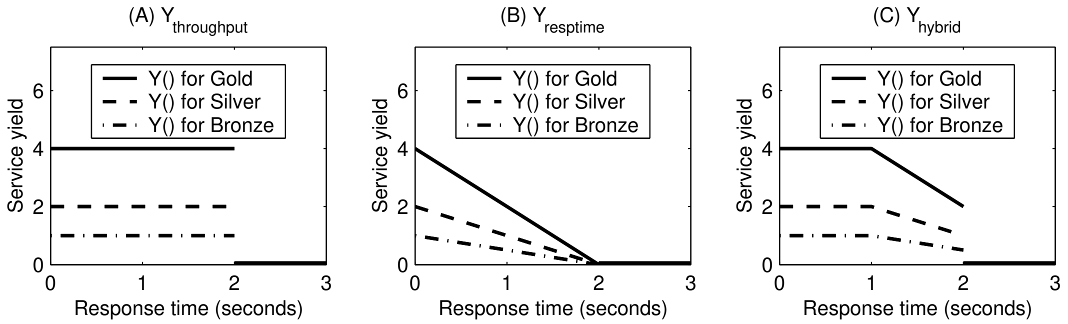

Figure 6: Service yield functions in evaluation workloads.

based on the same service type and thus have the same average resource consumption. The second service we constructed for the evaluation is designed to have different resource consumption for each service class, representing services differentiated on their types. This service, we call *Micro-benchmark*, is based on a CPU-spinning micro-benchmark. It contains three service classes with the same yield functions as the Differentiated Search service. The mean service times of the three classes are 400 ms, 200 ms, and 100 ms respectively. We use Poisson process arrivals and exponentially distributed service times for the Micro-benchmark service. Several previous studies on Internet connections and workstation clusters suggested that both the HTTP inter-arrival time distribution and the service time distribution exhibit high variance, thus are better modeled by Lognormal, Weibull, or Pareto distributions [15, 21]. We choose exponentially-distributed arrival intervals and service times for the following reasons. First, a primary cause for the high variance of HTTP arrival intervals is the proximity of the HTTP request for the main page and subsequent requests for embedded objects or images. However, if we only consider resource-intensive service requests which requires dynamic content generation, HTTP requests for embedded objects are not counted. Secondly, the service time distribution tends to have a low variance for services of the same type. Our analysis on the Ask Jeeves trace shows that those distributions have similar variances as an exponentially distributed sample would have.

5.2 Evaluation on Node-level Scheduling and Service Differentiation

In this section, we study the performance of four service scheduling policies (EDF, YID, Greedy and Adaptive) and their impact on service differentiation. The performance metric we use in this study is *LossPercent* [22], which is computed as

$$\text{LossPercent} = \frac{\text{OfferedYield} - \text{RealizedYield}}{\text{OfferedYield}} \times 100\%$$

OfferedYield is the aggregated full yield of all arrived requests and *RealizedYield* is the amount of yield realized by the system. We choose the loss percentage as the performance metric because this metric is effective in illustrating performance difference in both system underload and overload situations. In comparison, the actual rate (aggregate service yield in this case) is not as illustrative as the loss percentage when the system load is below the saturation point.

Figure 8 shows the performance of scheduling policies on Differentiated Search with 16 replicated servers. The experiments were conducted for all three forms of yield functions: $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, and $Y_{\text{hybrid}}()$. Figure 9 shows the performance of scheduling policies on Micro-benchmark with 16 servers. Only the result for yield functions in $Y_{\text{hybrid}}()$ form is shown to save space. In each case, we show the performance results with a varying arrival demand of up to 200% of the available resources. The demand level cannot simply be the mean arrival rate times the mean service time due to various system overhead. We probe the maximum arrival rate such that more than 95% of all requests are completed within the deadline under EDF scheduling. Then we consider the request demand is 100% at this arrival rate. The desired demand level is then achieved by scaling the request arrival intervals. The performance results are separated into the under-load (arrival demand $\leq$ 100%) and overload (arrival demand $\geq$ 100%) situations. We employ no minimum resource guarantee for both services to better illustrate the comparison on the aggregate yield. From these results, we observe that YID outperforms Greedy by up to 49% when the system is underloaded and Greedy performs up to 39% better during system overload. The Adaptive policy is able to dynamically switch between YID and Greedy policies to

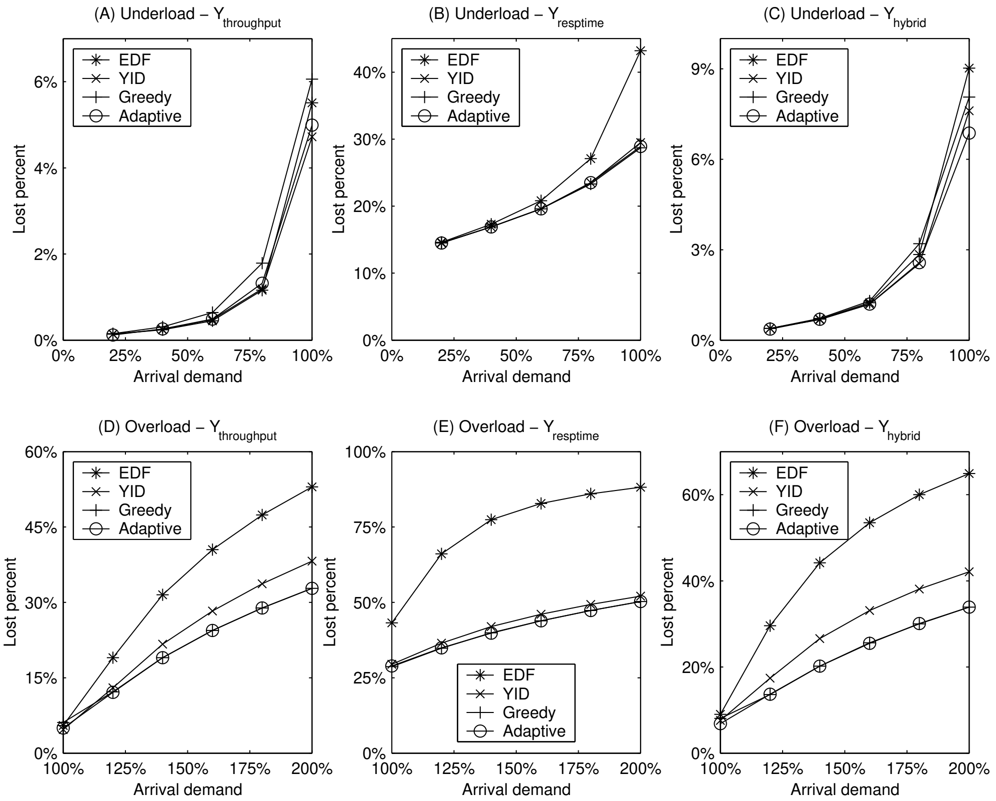

Figure 8: Performance of scheduling policies on Differentiated Search (16 servers).

achieve good performance on all studied load levels.

To further understand the performance difference among the scheduling policies and the impact on service differentiation, Figure 10 lists the per-class performance breakdown for Differentiated Search service with $Y_{\text{hybrid}}()$ yield functions under 200% arrival demand. For the per-class response time, we show both the mean values and the 95th percentile values. We choose a high arrival demand (200%) for this experiment because service differentiation is more critical at higher load. Extraordinary events can cause such severe system overload and shorter-term spikes can be more widespread in practice. From Figure 10, we observe that all four policies achieve similar aggregate throughput, however, Greedy and Adaptive policies complete more requests of higher-priority classes, representing more efficient resource utilization. In terms of the mean response time, Greedy and Adaptive policies complete requests with shorter mean response time, representing better quality for completed requests.

5.3 Evaluation on Request Distribution across Replicated Servers

Figure 11 illustrates our evaluation results on two request distribution schemes: class-aware load balancing (used in Neptune) and server partitioning. For each service, we show the aggregate service yield of up to 16 replicated servers under slight under-load (75% demand), slight overload (125% demand), and severe overload (200% demand). The Adaptive scheduling policy is used in each server for those experiments. The aggregate yield shown in Figure 11 is normalized to the Neptune yield under 200% arrival demand. Our result shows that both schemes exhibit good scalability, which is attributed to our underlying load balancing strategy, the random-polling policy that discards slow-responding polls [34]. In comparison, Neptune produces up to 6% more yield than server partitioning under high demand. This is because Neptune allows the whole cluster-wide load balancing for all service classes while server partitioning restricts the scope of load balancing to the specific server partition for the corresponding service class, which affects the load balancing performance.

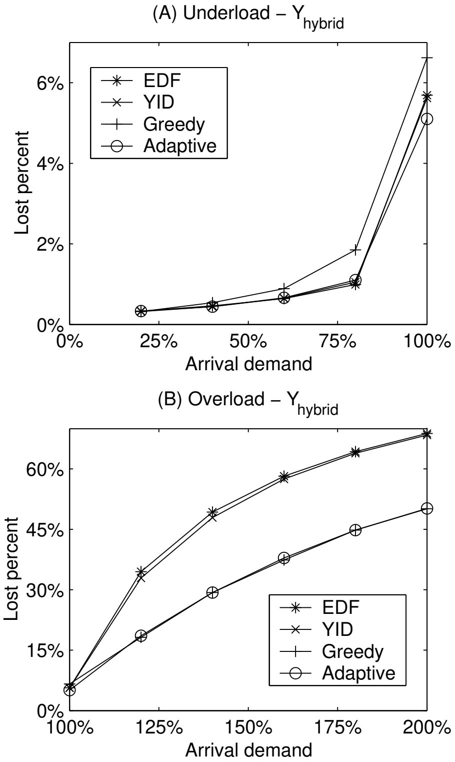

Figure 9: Performance of scheduling policies on Microbenchmark (16 servers).

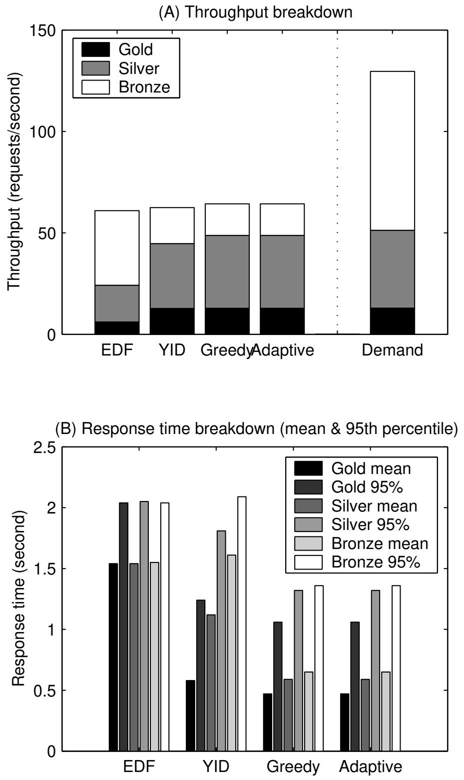

Figure 10: Per-class performance breakdown of Differentiated Search at 200% arrival demand.

5.4 Service Differentiation during Demand Spikes and Server Failures

In this section, we study the service differentiation during demand spikes and server failures. We use the Differentiated Search service with 20% resource guarantee for each class. In order to produce constantly controllable demand levels, we altered this service to generate fixed interval request arrivals. Figure 12 illustrates the system behavior of such a service under Neptune and server partitioning approaches in a 16-server configuration. For each service class, we show the resource demand and the resource allocation, measured in two-second intervals, over a 300-second period.

Initially the total demand is 100% of the available resources, with 10%, 30%, and 60% of which belong to Gold, Silver, and Bronze class respectively. Then there is a demand spike for the Silver class between time 50 and time 150. We observe that Neptune promptly responds to the demand spike by allocating more resources to meet high-priority Silver class demand and dropping some low-priority Bronze class requests. This shift stops when Bronze class resource allocation drops to around 20% of total system resources, which is its guaranteed share. We also see the resource allocations for Silver and Bronze class quickly stabilize when they reach new allocation levels. In comparison, the server partitioning scheme responds to this demand spike in a slower pace because it cannot adjust to immediate demand changes until the next allocation interval. We also observe that the resource allocation for the highest-priority Gold class is isolated from this demand spike under both schemes.

At time 200, one server (allocated to the Gold class under server partitioning) fails and it recovers at time 250. Immediately after the server failure, we see a deep drop of Gold class resource allocation for about 10 seconds under server partitioning. This is again because it cannot adjust to immediate resource change until the next allocation interval. In comparison, Neptune exhibits much smoother behavior because losing any one server results in a proportional loss of resources for each class. Also note that the loss of a server reduces the available resources, which increases the relative demand to the available resources. This effectively results in another resource shortage. The system copes with it by maintaining enough allocation to Gold and Silver classes

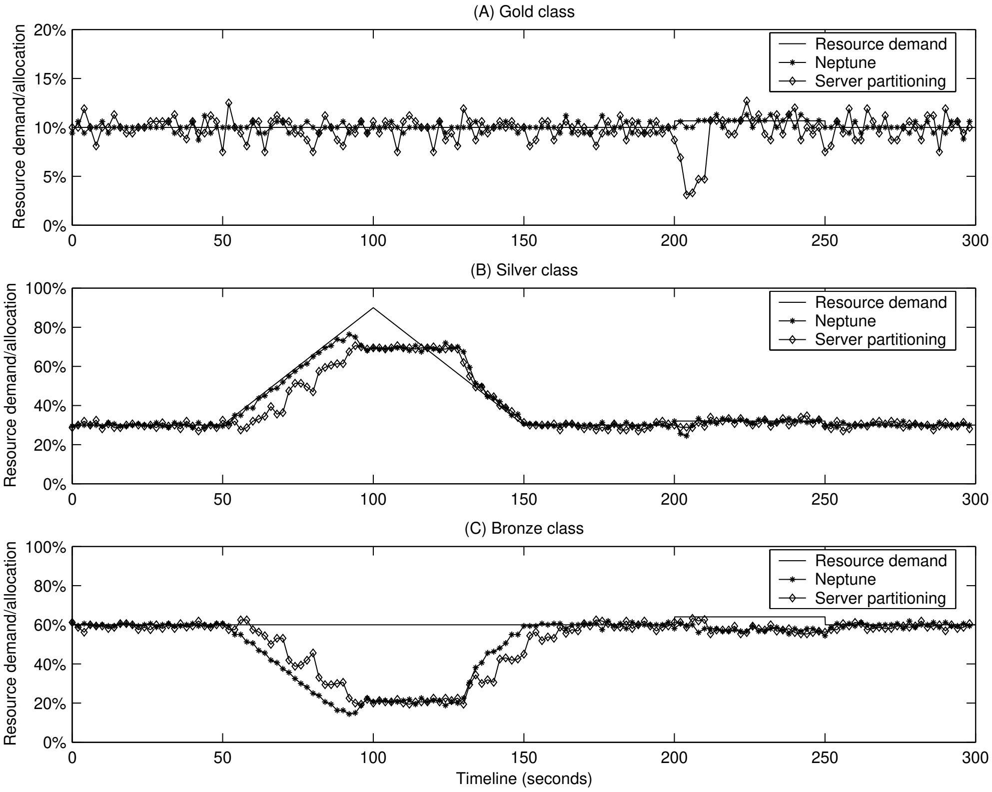

Figure 12: System behavior during demand spike and server failure with 16 servers. Differentiated Search with 20% resource guarantee for each class is used. One server (allocated to the Gold class under server partitioning) fails at time 200 and it recovers at time 250.

while dropping some Bronze class requests.

6 Related Work

Software infrastructure for clustered services. Previous studies have addressed the scalability and availability issues in providing software infrastructure for cluster-based network services [17, 20, 35]. In particular, TACC employs a two-tier architecture in which the service components called "workers" run on different back-ends while accesses to workers are controlled by front-ends [17]. Our work in this paper complements these studies by proposing an integrated resource management framework that addresses quality specification, efficient resource utilization under quality constraints, and service differentiation support. A number of recent works have studied replication support for services with frequent updates on persistent service data [19, 33, 35]. The focus of these studies is to support replica consistency in addition to the existing requirements of scalability and availability. The resource management framework proposed in this paper has only been evaluated with read-only workload. The impact of update-intensive workload remains to be addressed in the future.

Quality-of-service support and service differentiation. The importance of providing QoS support and service differentiation has been recognized in the networking community and the focuses of these studies is network bandwidth allocation and packet delay [26, 37]. The methods for ensuring bandwidth usage include delaying or dropping user requests [12, 27, 32] or reducing service qualities [1, 9]. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 2, 6, 7, 8, 27, 39] or clustered systems serving static HTTP content [3, 32]. In comparison, Neptune focuses on achieving efficient resource utilization and providing service differentiation for cluster-based services in which contents are dynamically generated and aggregated. Recent advances in OS research have developed approaches to provide QoS support at OS kernel level [6, 8, 13, 29, 36, 39]. Our work can be enhanced by those studies to support hard QoS

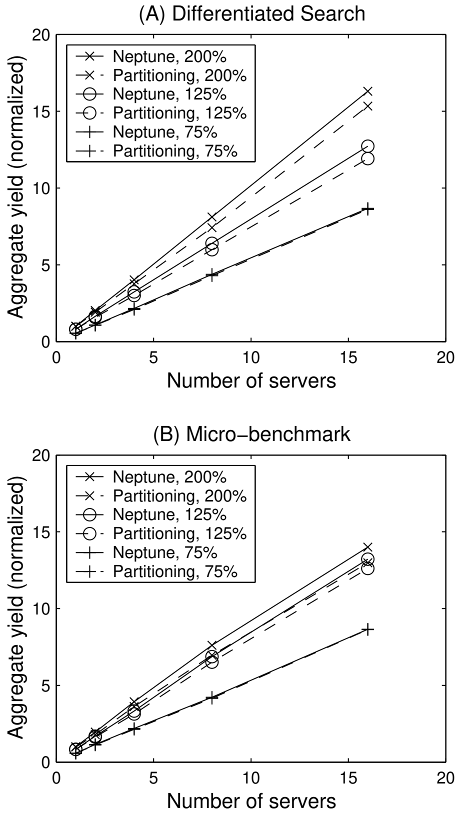

Figure 11: Performance and scalability of request distribution schemes.

guarantees and service differentiation at finer granularities.

The concept of service quality in this resource management framework refers to only the service response time. Service response time is important for many applications such that the proposed techniques can be widely applied. However, we acknowledge that service quality can have various application-specific additional dimensions. For instance, the partial failure in a partitioned search database results in a loss of harvest [16]. Further work is needed to address additional application-specific service qualities.

Resource management for clustered services. A large body of work has been done in request distribution and resource management for cluster-based server systems [3, 4, 10, 31, 38, 41]. In particular, demand-driven service differentiation (DDSD) provides a dynamic server partitioning approach to differentiating services from different service classes [41]. Similar to a few other studies [3, 10], DDSD supports service differentiation in the aggregate allocation for each service class. In comparison, this paper presents a decentralized architecture to achieve scalability while deploying quality-aware resource management.

Locality-aware request distribution. Previous study has proposed locality-aware request distribution (LARD) to exploit application-level data locality for Web server clusters [31]. Our work does not explicitly consider data locality because many applications are not locality-sensitive. For example, the critical working set in many Ask Jeeves service components are designed to fit into the system memory. Over-emphasizing on application-level service characteristics may thus limit the applicability of our framework. Nonetheless, it will be a valuable future work to incorporate locality-aware heuristics into our cluster-level request distribution and evaluate its impact on various applications.

Service scheduling. Deadline scheduling, proportional-share resource scheduling, and value-based scheduling have been studied in both real-time systems and general-purpose operating systems [7, 22, 23, 24, 30, 36, 40]. Client request rates for Internet services tend to be bursty and fluctuate dramatically from time to time [5, 10, 11]. Delivering satisfactory user experience is important during load spikes. Based on an adaptive scheduling approach and a resource consumption estimation scheme, the service scheduling in Neptune strives to achieve efficient resource utilization under quality constraints and provide service differentiation.

7 Concluding Remarks

This paper presents the design and implementation of an integrated resource management framework for cluster-based network services. This framework is flexible in allowing service providers to express desired service qualities based on the service response time. At the cluster level, a scalable decentralized request distribution architecture ensures prompt and smooth response to service demand spikes and server failures. Inside each node, an adaptive multi-queue scheduling scheme is employed to achieve efficient resource utilization under quality constraints and provide service differentiation. Our trace-driven evaluations show that the proposed techniques can efficiently utilize system resources under quality constraints and provide service differentiation. Comparing with a previously proposed dynamic server partitioning approach, the evaluations also show that our system responds more promptly to demand spikes and behaves more smoothly during server failures.

Acknowledgment. This work was supported in part by NSF CCR-9702640, ACIR-0082666 and 0086061. We would like to thank Anurag Acharya, Josep Blanquer, Apostolos Gerasoulis, Klaus Schauser, our shepherd Jim Gray, and the anonymous referees for their valuable comments and help.

References

[1] T. F. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Proc. of SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, June 1998.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proc. of the 2000 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, CA, June 2000.

[4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Services. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[5] Ask jeeves search. http://www.ask.com.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[7] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, September 1999.

[8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of USENIX Annual Technical Conf.*, pages 235–246, Orleans, LA, June 1998.

[9] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In *Proc. of IEEE INFOCOM'2000*, Tel-Aviv, Israel, March 2000.

[10] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[11] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[12] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proc. of ACM SIGCOMM'99*, pages 109–120, Cambridge, MA, August 1999.

[13] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[14] J. Postel Ed. Transmission Control Protocol Specification. SRI International, Menlo Park, CA, September 1981. RFC-793.

[15] A. Feldmann. Characteristics of TCP Connection Arrivals. Technical report, AT&T Labs Research, 1998.

[16] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proc. of HotOS-VII*, Rio Rico, AZ, March 1999.

[17] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint Malo, October 1997.

[18] Google search. http://www.google.com.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[20] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.*, Monterey, CA, June 1999.

[21] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load

Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[22] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *VLDB Journal*, 2:117–152, 1993.

[23] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. In *Proc. of the Tenth IEEE Real-Time System Symposium*, pages 144–153, Santa Monica, CA, 1989.

[24] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malo, France, October 1997.

[25] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, March 1972.

[26] J. Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *ACM Computer Communication Review*, 23(1):6–15, 1993.

[27] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proc. of IEEE INFOCOM'2000*, pages 651–659, Tel-Aviv, Israel, March 2000.

[28] Z. Liu, M. S. Squillante, and J. L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proc. of 3rd ACM Conference on Electronic Commerce*, pages 14–17, Tampa, FL, October 2001.

[29] J. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, January 1996.

[30] S. Nagy and A. Bestavros. Admission Control for Soft-Deadline Transactions in ACCORD. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 160–165, Montreal, Canada, June 1997.

[31] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.

[32] R. Pandey, J. F. Barnes, and R. Olsson. Supporting Quality of Service in HTTP Servers. In *Proc. of 17th ACM Symposium on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998.

[33] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Charleston, SC, December 1999.

[34] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel & Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[35] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 197–208, San Francisco, CA, March 2001.

[36] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proc. of 3rd USENIX Operating Systems Design and Implementation Symposium*, New Orleans, LA, February 1999.

[37] I. Stoica and H. Zhang. LIRA: An Approach for Service Differentiation in the Internet. In *Proc. of Nossdav*, June 1998.

[38] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[39] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proc. of USENIX Annual Technical Conf.*, Boston, MA, June 2001.

[40] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of USENIX Operating Systems Design and Implementation Symposium*, pages 1–11, Monterey, CA, November 1994.

[41] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation for Cluster-based Network Servers. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.