# QoS for Internet Services – Done Right.

*Josep M. Blanquer, Antoni Batchelli, Klaus Schauser and Rich Wolski*
Department of Computer Science, University of California Santa Barbara
{blanquer,tbatchel,schauser,rich}@cs.ucsb.edu

## Abstract

In this paper we argue that the best approach to providing Quality of Service (QoS) guarantees to current Internet services is to use admission control and traffic shaping techniques at the entrance points of Internet hosting sites. We propose a black-box approach that does not require knowledge, instrumentation, or modification of the system (hardware and software) that implements the services provided by the site.

We maintain that such a non-intrusive QoS solution achieves better resource utilization, has lower cost, and is more flexible than the current approaches of physical partitioning and hardware over-provisioning. Furthermore, we contend that our solution is easier to deploy, less complex to implement, and easier to maintain than more intrusive approaches which embed the QoS logic into the operating system, distributed middleware, or application code. We demonstrate empirically that despite being decoupled from the internal mechanisms implementing the site, a black-box approach provides effective response times and capacity guarantees.

## 1   Introduction

With the increasing importance of Internet services, it is imperative for companies relying on web-based technology to offer (and potentially guarantee) predictable, consistent, as well as differentiated quality of service to their consumers. For example, a search engine such as Google may want to guarantee a different service quality for the results served to America On-Line (AOL) than the quality it can guarantee to Stanford University searches. 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 to all interested clients. Implementing service quality guarantees scalably in such a distributed setting is a difficult challenge.

Traditional approaches to solving this QoS challenge treat the problem as a capacity planning matter and rely on over-provisioning the resources and on physically partitioning groups of cluster nodes for different classes of service. Unfortunately, the necessity to handle enormous and unpredictable fluctuations in load results in these techniques suffering from high cost (enough resource must be available in each partition to handle load spikes) and low resource utilization (the extra resources are idle between spikes). More-over, such a static approach does not offer much flexibility, requiring hardware reconfiguration for any modification of the QoS policy or change in the recurring load pattern. Furthermore, although these hardware-based approaches can certainly improve the quality of the service, they cannot provide QoS guarantees unless each of the partitions are always kept from being overloaded.

As a result, software-based approaches have been suggested that embed the QoS logic into the internals of the operating system [3, 5, 16, 2], distributed middleware [19, 14], or application code [1, 4, 17, 9] running on the cluster. Operating System 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 important to the clients. However the majority of existing software approaches offer guarantees within the scope of a single machine or for an individual application, and fail to provide global service quality throughout the site. Most current Internet sites are composed of a myriad of different hardware and software platforms which are constantly evolving and changing. The largest drawback to software-based approaches is the high cost and complexity of reprogramming, maintaining, and extending the entirety of the complex software system such that it can provide QoS guarantees for all hosted services.

Moreover, many components of a service are commonly third-party, proprietary software (e.g., Commercial Databases, Application Servers, etc) for which the source code is not available. Approaches that rely on embedding QoS support directly into each application are difficult to implement in these cases. When the applications themselves have embedded QoS support, it is often a mechanism that is unique to a particular application which makes enabling QoS and interoperability very difficult.

We propose to provide QoS through the use of traffic shaping and admission control techniques at the entrance of Internet sites. Our approach treats the cluster and the services it is hosting as an unknown "black-box" system and uses feedback-driven techniques to dynamically control how each of the requests from the clients must be forwarded into the cluster. We strongly believe that this approach attains the best of both hardware and software worlds, being
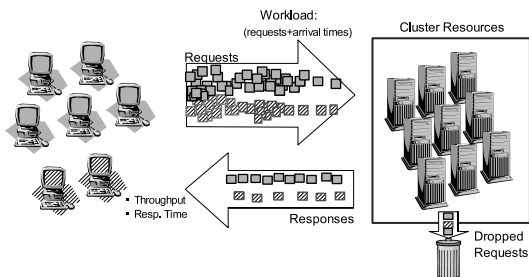
Figure 1: System Model for Internet Services

a uniform solution that does not require hardware reconfiguration or software reprogramming while still ensuring effective QoS guarantees for Internet services.

## 2 The QoS Challenge

In this section, we provide some background on QoS for Internet services, and define the problem we are trying to solve. We model Internet services (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 (e.g., URL, protocol, port, etc.) and client identity (e.g., source IP address, cookies, etc.). A *QoS class* describes the predefined quality to be enforced for a particular service class and the collection of all existing QoS classes forms a *QoS policy*.

We define a QoS class as a tuple with three quantities: guaranteed *throughput*, *computing requirements* to fulfill a request, and maximum *response time* allowed. For example, a QoS class for a typical e-Commerce site could specify a minimum throughput guarantee of 200 req/s, requiring a computation of 10ms each, and a maximum response time of 500ms. Throughput and response times are commonly expressed using percentiles or averages that must be ensured over time intervals that are much longer than the specified computing requirements.

We view the QoS challenge as the ability to enforce a feasible set of QoS guarantees for a given cluster under all input load conditions. A set of guarantees is feasible if the cluster can honor them, without using any QoS mechanism, when the incoming traffic for each class is below its guaranteed throughput. In other words we require the cluster to be appropriately provisioned for the given QoS policy. We note that the function of a QoS system is to enforce a set of guarantees. It cannot resolve resource bottleneck problems if the site implements insufficient hardware provisioning.
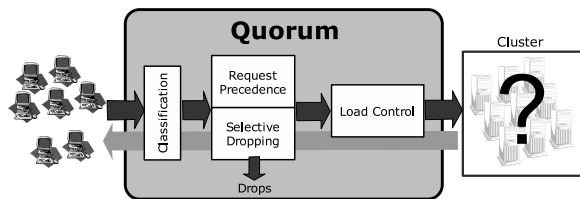


Figure 2: The Architecture of Quorum

In addition to ensuring quality guarantees, any solution to the QoS problem should also have other desirable properties. A solution should achieve good resource utilization (*i.e. be efficient*), not require any internal cluster modification (*non-intrusive*), support a broad range of QoS needs (*comprehensive*), support varying workloads and cluster changes (*adaptive*), and degrade gracefully under overload (*robust*). We strongly believe that the architectural characteristics of our solution are able to attain all of these desired properties while meeting the QoS challenge in scalable cluster environments.

## 3 The Quorum Approach

Our approach to QoS for Internet services consists of deploying a single QoS engine at the entrance of the Internet sites which contains the system-wide QoS policy that needs to be enforced. Figure 2 depicts the architecture of *Quorum*, a QoS engine consisting of four modules, that follows the approach presented in this paper. The *Classification* module categorizes the intercepted requests from the clients into one of the service classes defined in the QoS policy. The *Load Control* module determines the pace (for the entire system and all client request streams) in which Quorum releases requests into the cluster. The *Request Precedence* module dictates the proportions in which requests of different classes are released to the cluster. The *Selective Dropping* module drops requests of a service class to avoid work accumulation and maintain responsiveness. The modules of Quorum are designed to address each of the four different functionalities we believe any QoS system must have. In the next sections we explain why these functions are necessary and provide more details on how they are built into our modules. We explicitly exclude the details of Classification since it is a well understood problem that has already been studied in the literature [10] and being extensively used by current sites in the form of firewalls and load-balancers.

### 3.1 Load Control

The primary function of *Load Control* is to prevent incoming traffic from overloading the internal resources of the cluster. This functionality is necessary because no quality guarantees can be enforced if the cluster is operating in an

overloaded state. The Load Control module externally controls the load in the cluster by appropriately either forwarding or holding the traffic received from the clients, according to current performance metrics measured at the output of the cluster.

Similar to TCP, our implementation uses a sliding window scheme which defines the maximum number of requests that can be outstanding at any time. The Quorum engine tries to increment successively the size of the window until the QoS class with the most restrictive response times approaches the limits defined by its guarantees.

## 3.2 Request Precedence

The function of *Request Precedence* is to virtually partition resources among each of the service classes. Capacity isolation is a necessary functionality that allows each service class to enjoy a minimum amount of guaranteed resources, 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.

Under Quorum, Request Precedence is implemented through the use of modified Weighted Fair Queuing [8, 13, 6] techniques that function at the request level. By factoring the expected computing requirements of each class into the fair queuing weights, Quorum transforms throughput guarantees into capacity guarantees. Capacity (or computing power) is a fungible metric that links output throughput and computing requirements in a way that an increment of one results in a decrement of the other. For example a capacity of 4000ms/s corresponds to 400req/s at a compute cost of 10ms/req, but also to 800req/s if the compute cost is only 5ms/req. Using this fungible metric, the Request Precedence safely protects the service classes even when one or more of them violate the expected computing requirements specified in the QoS class.

## 3.3 Selective Dropping

The function of *Selective Dropping* is to discard the excessive traffic received for a service class in the situations where it becomes overloaded. A service class becomes *overloaded* when its guaranteed capacity it is not enough to fulfill the totality of its incoming traffic. A dropping module is necessary to prevent large delays from occurring in overloaded situations. This module observes each of the queues of the engine and discards the requests that have been sitting in the queue for too long.

In Quorum, 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 Quorum to absorb safely peaks of traffic during transient overload conditions without violating the response time guarantees.

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

# 4 Preliminary Evaluation

In this section we show that Quorum as a black-box software technique can provide effective response time and throughput guarantees for Internet-based services. We empirically show how our approach outperforms the current hardware-based techniques in terms of resource utilization, cost and flexibility. We also present arguments that substantiate why our non-intrusive approach is better suited than other software approaches to managing the current complexity associated with the implementation of Internet services.

To show the effectiveness of our technique, we define an emulated QoS case that is difficult to address and observe how each approach performs under the same conditions. Our experiments compare four alternatives: a shared cluster with no QoS control (*No Control*), a cluster with one physical partition for each service class (*Physical Partitioning*), a physically partitioned cluster that has each partition overprovisioned (*Overprovisioning*) and a shared cluster with our module at the entrance (*Quorum*). We perform the experiments using a 12-CPU cluster running the Tomcat [15] web server and servlet engine. In the oveprovisioning case we use 36 CPUs. We generate client requests by replaying Web server traces of a real service provider [7] so that the arrival times exhibit ill-behaved characteristics of current Internet services.

We examine a common QoS scenario in which a large number of clients compete for services that are hosted in the same physical facility. In our scenario we emulate three different types of Internet services of different computational complexity and we set up one class of clients to generate service demands that far exceed the capacity that they have guaranteed. The three emulated services[1] are designed to approximate the complexity of a typical *e-Commerce*, *Stock* trading and *Search* service respectively. Table 1 has further details of the experiment, including the throughput and response time guarantees selected for each service.

---

[1]The services are emulated by using a certain amount of CPU through a loop, and generating typical page sizes.

| Service Class | QoS Guarantees | | | Experimental Workload | |
|---|---|---|---|---|---|
| | 90th percentile Resp. Time | Average Throughput | Avg Compute Requirement | Average Input Traffic | Service Status |
| **e-Commerce** | 150ms | 350 req/s | 10 ms/req | 170 req/s | Not Overloaded |
| **Stocks** | 500ms | 175 req/s | 30 ms/req | 331 req/s | Overloaded |
| **Search** | 600ms | 18 req/s | 100 ms/req | 12.4 req/s | Not Overloaded |

Table 1: QoS Guarantees and Traffic Workload of the Experiment



Figure 3: Throughput Results



Figure 4: Response Time Results
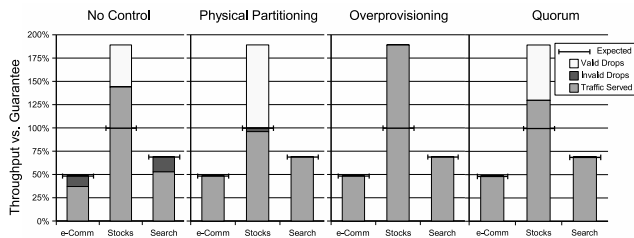
## 4.1 Throughput Guarantees

We begin by comparing the impact on throughput for each of the four approaches. Figure 3 presents the total amount of traffic both served and dropped. The scale for each class is normalized to its guaranteed throughput (i.e., 100% for the e-Commerce class corresponds to 350 req/s). Horizontal marks delimit the amount of traffic expected to be served if the guarantees were met. In the cases where a service class receives less incoming traffic than its guaranteed throughput (i.e., e-Commerce and Search) we expect *all* requests to be served.

The results obtained show that the amount of traffic served when there is *no* QoS *control* is directly dependent on the input demands, making it impossible to provide any output guarantees. In this case we can see that the dominance of Stocks traffic provokes drops in the e-Commerce and Search classes even though these classes only use 48% and 69% respectively of their legitimate guarantee (invalid drops). When using physical partitioning, the system always serves the expected amount of traffic for each class and only drops requests when incoming traffic demands exceed the available capacity of the partition (valid drops in the Stocks class). What it is not visually striking from this graph is that the Stocks partition looses 4% of its guaranteed traffic. This is due to the detrimental performance impact of having the partition operating at an extreme overload. In the case of overprovisioning there are enough resources to serve all the traffic that is received. Our results show that despite being a shared cluster, Quorum not only serves the amount of expected traffic for each service class, but also reassigns the un-utilized capacity to accommodate more clients accessing the Stocks service. Note that Quorum gives an extra 30% of traffic to the Stocks clients without affecting the performance (i.e., dropping requests) of the e-Commerce and Search services.
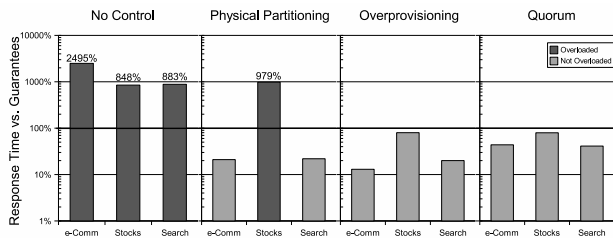
## 4.2 Response Times

Next we analyze the impact on response times. Figure 4 shows the 90th percentile of response times obtained by each of the three alternatives. Response Times are normalized to the guarantees and presented in logarithmic scale for better visual comparison. Achieving a response time over 100% denotes an excessive delay, hence a failure to fulfill the QoS guarantee. As expected, when there is no QoS control the servers become completely overloaded, accumulating large amounts of traffic which results in all service classes experiencing unboundedly long delays. When using physical partitioning, the classes that are not overloaded (i.e., e-Commerce and Search) successfully meet their response time requirements. However we see that the overloaded Stocks partition experiences an average delay that is almost 10 times higher than the maximum allowed. Quorum and Overprovisioning are the only two approaches that can successfully provide response time guarantees independently for each of the classes, regardless of how much incoming traffic they receive. However, we will see in the next section that achieving such guarantees through the use of overprovisioning comes at a very high cost.

## 4.3 Resource Utilization

Figure 5 shows both the cost and resource utilization of each of the approaches. The utilization percentages are calculated as the available computational power divided by the amount of work processed. Presented results are normalized to the maximum observed performance of the cluster for the tested workload as determined by an offline analysis. Our results show that while overprovisioning must maintain a low utilization of the cluster (i.e., 43%) to achieve fast service times, Quorum can accomplish the same goals at 99% resource utilization. It is interesting to observe that
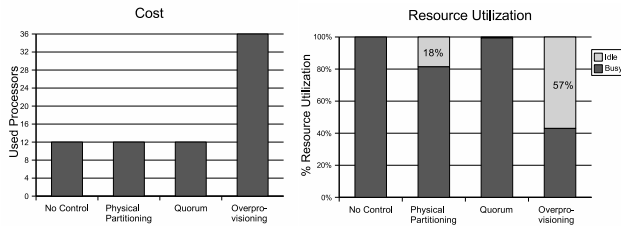
Figure 5: Cost and Resource Utilization Results

despite tripling the size of the cluster, the resulting response times for the Stocks partition are only 1% faster than those achieved by Quorum (i.e., 477ms vs. 484ms).

## 4.4 Alternative Software Approaches

The existing alternatives to hardware approaches are either based on *instrumenting* the internal software to actively participate in QoS decisions, or based on a solution that requires a precise *knowledge* of the internal operation of the site. In this section we briefly introduce such approaches and provide arguments for Quorum as a more flexible solution than the current state of the practice.

Software instrumentation can be implemented at the operating system [3, 5, 16, 2], middleware [19, 14, 18] or application code [1, 4, 17], and commonly requires close cooperation between them. While the majority of the existing solutions can offer QoS guarantees within the scope of a single machine, only a few are designed to provide cluster-wide QoS. For example, Cluster Reserves [2] requires a tailored OS [3] running at each node of the cluster and needs a centralized controller to constantly modify the resource allocation at the nodes based on reported usage statistics. Other approaches such as Neptune [14] or ControlWare [18] provide QoS in the form of a middleware infrastructure which applications can utilize to implement distributed QoS coordination. However, implementing these techniques requires installing a tailored operating system at each of the internal nodes, tuning and deploying a complex middleware infrastructure, and even modifying each service application to interact with the new QoS primitives. We believe that such intrusive modifications are not always viable due to lack of application source code, incompatibilities between the applications and the patched operating system, or site management policies. Even in the cases where modifications are possible, the magnitude of the changes and their maintenance cost would make the resulting system have a much higher implementation complexity as well as less flexibility than deploying Quorum at the entrance of the site.

Few alternatives exist that do not require internal modifications. However these approaches rely on having a precise knowledge of the internals of the site [11, 12] or depend on deriving its service behavior by means of static application profiling [9]. For example, many of the commercially available load-balancer solutions can be highly tuned for a specific site configuration such that system overload can be prevented. Other work such as Gatekeeper [9] proposes a proxy system, much like Quorum, that implements admission control for e-commerce applications. Although Gatekeeper can be considered an external system, it requires knowing the total available capacity of the cluster for the tested workload (which must be done offline) and relies on extensive profiling of the service application to avoid cluster overload and to reduce service times. Although similar in nature to Quorum, these approaches suffer from poor flexibility, requiring reconfiguration, retuning or re-profiling the applications at every hardware upgrade and for each addition of a new service. Moreover, these approaches are not designed to provide QoS guarantees but rather, they are designed to improve the overall performance of the cluster.

In conclusion, with the current heterogeneity, sophistication and rapid evolution of current Internet services, we argue that any intrusive software approach to cluster-wide QoS has a higher implementation complexity and lower flexibility than an external technique such as Quorum.

## 5 Conclusions

In this position paper, we propose a novel technique for providing QoS for Internet services. We have experimentally shown the benefits of our external technique versus hardware-based approaches and have given arguments on the advantages with respect to intrusive software approaches. We have also shown that despite what it may seem, an external black-box technique can achieve tight QoS guarantees even when the internal cluster is fully shared by different services. We firmly believe that the right approach to implement QoS for Internet services is to use admission control and traffic shaping techniques at the entrance points of Internet hosting sites.

## References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, June 2000.

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[4] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, September 1999.

[5] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Özden, and A. Silberschatz. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the First FreeBSD Conference*, Berkeley, California, Oct. 1999.

[6] J. Blanquer and B. Özden. Fair Queuing for Aggregated Multiple Links. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[7] Clarknet. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[8] A. Demers, S. Keshav, and S. Shenker. Design and Simulation of a Fair Queuing Algorithm. In *Proceedings of the ACM SIGCOMM*, Austin, Texas, September 1989.

[9] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th International World Wide Web Conference*, New York City, NY, USA 2004.

[10] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network Special Issue*, March 2001.

[11] Netscaler Request Switching Technology. http://www.netscaler.com.

[12] Packeteer Traffic Management http://www.packeteer.com.

[13] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks-the Single Node Case. *IEEE/ACM Transactions on Networking*, June 1993.

[14] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[15] Apache Tomcat server. http://jakarta.apache.org/tomcat.

[16] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[17] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct 2001.

[18] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, Providence, Rhode Island, May 2003.

[19] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of the IEEE INFOCOM*, Anchorage, Alaska, April 2001.