QoS Provisioning with qContracts in Web and Multimedia Servers

Tarek F. Abdelzaher
Department of Computer Science
The University of Virginia
Charlottesville, VA 22903
zaher@cs.virginia.edu

Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109
kgshin@eecs.umich.edu

Abstract

The advent of performance-critical services such as online brokerage and e-commerce, as well as QoS-sensitive services such as streaming multimedia, makes existing FIFO servers incapable of meeting application QoS requirements. Re-designing server code to support QoS provisioning, on the other hand, is costly and time-consuming. To remedy this problem, we propose a new QoS-provisioning approach that does not require modification of server and OS code. We develop a middleware, called qContracts, that can be transparently interposed between the server process and the operating system to achieve performance differentiation and soft QoS guarantees. The middleware enables reuse of existing legacy software in QoS-sensitive contexts, and off-loads QoS management concerns from future real-time service programmers. As an example, we show how the Apache [9] web server is endowed with QoS support using qContracts on UNIX. Experimental results show the efficacy of the middleware in achieving the contracted QoS, while imposing less than 1% overhead.

Index Terms — *real-time services, QoS provisioning, middleware*

1 Introduction

This paper presents a middleware layer, called the qContracts that exports a novel programming abstraction for building future performance-assured services. Programming with qContracts allows creating, manipulating and terminating QoS contracts with clients or client categories to achieve performance guarantees. The middleware enforces the contracted QoS on behalf of the service programmer, thus off-loading a substantial responsibility from designers of real-time services. Adoption of new, more convenient, programming abstractions has potential to reduce future programming efforts and development costs of QoS-sensitive software. However, new abstractions and APIs raise questions regarding their utility for legacy code. In order to address this issue as well, we describe how to use our QoS extensions transparently within the socket library, while exporting to the server a standard socket API.

To demonstrate the relevance of our work to existing practical applications, we motivate our middleware in the context of performance-assured web and multimedia services. Contemporary web and multimedia servers treat all clients equally and serve them on a first-come first-served basis. A server saturated by user requests will become non-responsive, thus creating the perception of service outage. Client requests for the server accumulate in the operating system in a FIFO queue (e.g., the server's TCP socket listen queue). In case of overload, the request queue overflows, exhibiting a drop-tail behavior that affects indiscriminately all clients irrespective of their importance.

The advent of critical applications such as e-commerce as well as QoS-sensitive applications such as multimedia necessitates improvement of traditional server design to support QoS provisioning and performance differentiation. QoS provisioning refers to providing guarantees on delivered QoS, such as service rate and bandwidth, to selected clients. Performance differentiation refers to giving preferential treatment to some clients over others. While operating system extensions such as capacity reserves [21] and QoS-aware sockets [35] would simplify the problem of developing QoS-sensitive services, in this paper we take an alternative approach of developing middleware support. The chief advantage of using middleware lies in its portability. Our middleware is usable on most UNIX variants with little or no customization. The disadvantage of a middleware approach lies in its insufficient control over OS resource allocation precluding hard real-time guarantees. We will describe in this paper how this insufficient control is countered to achieve "approximate" QoS guarantees suitable for a large class of soft real-time applications, such as web services and e-commerce. We will show that the cost of the middleware layer is insignificant.

QoS provisioning must account for platform speed in order to perform appropriate resource allocation and management. Characterization of platform speed and application requirements is generally a non-trivial process. This process must be repeated every time the platform or software is upgraded. The qContracts middleware saves the cost of repeated manual profiling by adapting to platform capacity and server resource demands using an automated self-profiling approach.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the target server architecture and the QoS-contract model. Section 4 elaborates on resource management to achieve transparent QoS provisioning and performance differentiation. Section 5 presents an example application. Section 6 discusses lessons learned, and Section 7 concludes the paper.

2 Related Work

Recently, QoS provisioning for multimedia and soft real-time applications has received considerable attention [4]. Since QoS provisioning is closely related to proper resource allocation and scheduling, much of research efforts have focused on operating system design. Research operating systems such as Rialto [15], Dreams [29], Nemesis [19], and Real-Time Mach [32], are just a few examples of kernels with real-time support.

Many kernel extensions have been proposed to provide real-time guarantees for QoS-sensitive applications. For example, capacity reserves [21] have been used in Mach to allocate processing capacity for multimedia applications [17], and flexible CPU reservation was used in Rialto for efficient scheduling of time-constrained independent activities [14]. Resource shares have been suggested as another mechanism for performance isolation. Examples include hierarchical CPU scheduling [11], proportional-share lottery and stride scheduling [34], and proportional-share allocation for time-shared systems [31]. Proportional-share scheduling has also been used to schedule system services [13]. Other notable multimedia-oriented kernel extensions include the SMART scheduler in Solaris [25], a real-time scheduler for Linux [30], and the concept of resource-centric kernels [26].

Multimedia applications are generally communication-intensive. Several communication-related architectures have therefore been proposed to support multimedia QoS guarantees. Examples include the QoS-A framework [6], the Heidelberg QoS model [33], V-net [8], NetWorld [7], the QoS-adaptation model of [3], COMETS' Extended Integrated Reference Model (XRM) [16], the OMEGA endpoint architecture [24], and the QoS Broker [23]. The design of QoS-sensitive operating system communication subsystems has been investigated in [20, 18, 35]. QoS-guaranteed protocol stack implementation in the user space has been proposed in [10, 19].

Recent research efforts also considered general adaptive resource management frameworks for real-time applications with elastic QoS constraints. The Q-RAM architecture [27] introduces QoS-sensitive near-optimal resource allocation algorithms for applications with multiple resource requirements and multiple QoS dimensions. FARA [28] presents a hierarchical adaptation model for complex real-time systems. An end-to-end QoS model similar to ours is presented in [12] in the context of a middleware approach to QoS management that requires application cooperation. The approach is extended in [5] to account for practical limitations such as inaccuracies in estimating application resource requirements.

In [2] we presented a QoS negotiation framework that attempts to maximize system utility and extended it to an architecture for OS communication subsystems [3]. In contrast to OS solutions, the work presented in this paper relies on portable middleware. We explicitly target server platforms. Since such platforms typically run a single application (the server), we do not deal with issues of trust among independent applications with conflicting requirements for which kernel-level enforcement solutions are more suitable. Unlike previous middleware approaches [5, 12] which introduced QoS extensions for applications with per-flow QoS constraints, we extend QoS specification and resource management mechanisms of [2] to traffic aggregates. These extensions are crucial to applications such as web-servers and differentiated services that deal with multiple aggregated flows. We pay particular attention to re-using legacy software. qContracts is designed for proper per-traffic-class QoS management even when used by legacy servers in which a single pool of identical same-priority threads (or processes) serves all traffic classes in FCFS order. Thus, while qContracts introduces new programming abstractions that encourage QoS-sensitive application design, it does not preclude using existing mainstream server code while providing QoS guarantees.

3 The Model

We target our middleware for contemporary mainstream web and multimedia servers. In a typical multithreaded server, worker threads execute a sequence of clients' requests. Each thread implements a loop that processes incoming requests and generates responses. We assume that requests and responses are read from, and written to, the machine's communication subsystem via the $read()$ and $write()$ socket library calls or their equivalents in non-UNIX operating systems. In our model, qContracts API may be called either directly by the server, or transparently to the server by instrumenting the aforementioned socket library calls. Worker threads are created either dynamically for incoming requests as they arrive, or *a priori* as a static pool. Our middleware makes no assumptions regarding the way server code assigns threads to requests or clients. The absence of restrictions on thread-to-client mapping separates our QoS-provisioning solution from earlier general mechanisms for performance isolation and QoS guarantees, such as [17, 14, 20, 2], that typically map threads to clients in a static fashion.

QoS requirements must be known *a priori* in order to provide performance assurances. We therefore require a separate *contract creation* stage to request QoS guarantees. This requirement does not entail changes to the current server code or service protocols. A process separate from the server, running on the server's machine, establishes QoS contracts on behalf of the server. We call it the *subscription agent*. In our current implementation, the contracts created by the subscription agent are stored in a configuration file accessed by the middleware.

For example, consider a web hosting service. Businesses outsource their web site management to the web-hosting service provider at which time a contract is created and stored by the subscription agent between the outsourcing party and the web-hosting service provider. The contract may specify QoS requirements such as the maximum aggregate request rate and average URL size the server will serve for the outsourced site. The middleware will apply admission control and, if successful, will allocate enough resources to satisfy the contracted QoS. As end-users request URLs from the server, the middleware will infer for each request the applicable contract from the requested site ID and will use resources allocated to that contract to serve the request. If the accessed site has no QoS contract, it is served at a lower priority. In the rest of this paper we use the term "client" to denote the party that signs the contract with the service provider and the term "user" to denote the party that connects to the server at run-time.

In our model, a QoS contract, Q_i, between the middleware and the client specifies a QoS-violation penalty V_i (which represents the "undesirability" of failure to deliver the contracted QoS), and declares one or more alternative QoS levels, $L_i[k]$, that the client accepts to receive from the server, and the reward or utility, $\$_i[k]$, of each level. In practice, contracts can be "prepared" by the service provider, giving the client a simple pre-arranged choice of service levels (and presumably the corresponding prices). Given the set of QoS contracts, the middleware allocates a QoS level k_i^* to each client, i, such that the sum of rewards $\$_i[k_i^*]$ for served clients less the sum of violation penalties V_i for rejected clients is maximum. We use the optimal psuedo-polynomial-time dynamic programming algorithm we presented in [3] for QoS-level selection that maximizes total reward. For web and multimedia services qContracts describes a QoS level, k, of contract Q_i by the following QoS parameters:

- *Service Rate, $R_i[k]$:* expressed as $M_i[k]$ *units of service* per specified period $P_i[k]$, i.e., $R_i[k] = M_i[k]/P_i[k]$). The units of service are arbitrary, but all contracts with a particular server must use the same units. Examples of service rate are: $M_i[k]$ served URLs per period (e.g., in web hosts), $M_i[k]$ served packets per period (e.g., in communication subsystems), or $M_i[k]$ served frames per period (e.g., in audio/video servers), the units of service being URLs, packets and frames in these examples, respectively.
- *Aggregate Data Bandwidth, $W_i[k]$:* specifies the aggregate bandwidth in bytes per second to be allocated for the client.

Service rate and data bandwidth are very useful QoS parameters because resource consumption at the end-system typically has two components: (i) a fixed per-unit-of-service overhead (such as per-packet protocol-processing cost), and (ii) a data-size-dependent overhead (such as data copying and transmission cost).

We do not deal with jitter and end-to-end response-time constraints in this paper, since their satisfaction depends on network support that cannot be guaranteed by the server alone. Next we discuss mechanisms for enforcing the desired QoS.

4 QoS Provisioning

Once the "optimal" QoS level has been selected for each contracted client, using the algorithm in [3], this level of QoS must be guaranteed regardless of the load imposed on the server by other clients. We call this requirement *performance isolation*. Non-contracted clients may be served at a lower priority whenever spare resources are available. We call this requirement *performance differentiation*. The novelty of qContracts lies in designing these two mechanisms in a way that can be implemented both *transparently to the server*, if desired, and *without operating system support*. Section 4.1 describes the resource-management mechanisms used for performance isolation. Section 4.2 describes transparent performance differentiation.

4.1 Performance Isolation

In order to guarantee contracted QoS, a resource-management mechanism must perform several functions. First, *QoS mapping* must be invoked to compute the resource requirements of each contract. *Contract admission control* must be performed to admit only as many contracts as would ensure that overload is avoided. *Per-contract policing* must be used to make sure that no QoS contract utilizes more resources than allocated to it and that malicious, misbehaving or greedy best-effort code serving a particular client does not infringe on the resources allocated to other clients of the same or higher priority. Finally, since the server may run on platforms of different speed and resources, *service profiling* is needed on each target platform to estimate its aggregate resource capacity. In what follows, we elaborate on the aforementioned components of QoS provisioning in qContracts.

4.1.1 QoS Mapping

QoS mapping is performed upon activation of a new contract. In general, QoS mapping is complicated due to the need to consider multiple resources when QoS specifications are translated into resource requirements. To simplify QoS mapping, we take advantage of the fact that our QoS contracts specify only QoS parameters with rate semantics. Unlike response times, for example, rates are constrained solely by the capacity of the bottleneck resource. To determine the bottleneck resource, we consider the resource consumption at both the end-system and the network. End-system resource consumption per service unit is approximated by a fixed overhead component plus another component that depends on served data size. In [1] we analyze this approximation in the context of web servers and show that it is accurate in describing server load. Thus, the time for the end-system to process a unit of service is $a + bx$, where a and b are constants, and x is the size of data served. Aggregating the end-system capacity consumed by processing a sequence of service units during some observation period, the consumed end-system utilization is $U = aR + bW$, where R is the service rate and W is the served byte-rate. The consumed network utilization, on the other hand, depends only on the byte rate and is therefore given by $U_n = cW$. The utilization of the bottleneck resource is $max(aR + bW, cW)$. Thus, to satisfy the QoS specification of contract Q_i, we logically allocate for the contract a bottleneck resource utilization budget equal to:

$$U_i = max(aM_i/P_i + bW_i,\ cW_i) \tag{1}$$

where M_i/P_i and W_i are the service rate and bandwidth parameters of the QoS level at which the contract is served. This logical allocation does not require OS calls. It simply quantifies the bottleneck resource capacity needed to satisfy the QoS requirements so that it is accounted for in contract admission control. Note, from Equation (1), that the bottleneck resource is the network if the average byte size per unit service $W_i/R_i > a/(c-b)$, where $R_i = M_i/P_i$, and is the end-system otherwise. We have seen that the bottleneck resource is largely independent of the individual client, but mainly is a function of the service platform. This is because the average size, W_i/R_i, of a unit of service does not vary considerably from one client to another. For example, the average video frame size does not depend considerably on the particular movie. Having computed the required bottleneck resource utilization, U_i, the identity of the bottleneck resource is no longer relevant, which simplifies admission control and policing.

4.1.2 Overload Protection and Admission Control

Overload is avoided by ensuring that $U = \sum_i U_i$ for all created contracts does not exceed 100%. When a new contract, Q_i, is created its required utilization, U_i, is computed using Equation (1) and added to U. If the new utilization sum exceeds 100%, the new contract is rejected. Note that keeping required utilization below 100% guarantees eventual completion of all periodic tasks within one common multiple of their periods. It does not guarantee completion of each task invocation by its period unless EDF scheduling is used. However, since we do not deal with hard deadlines in our model, we do not require EDF scheduling. For web applications, the aforementioned utilization condition is sufficient to ensure a small response time on average compared to the bandwidth of human perception.

4.1.3 Policing

At the beginning of every period P_i, an execution budget of P_iU_i is allocated by qContracts for each admitted contract Q_i, where U_i is given by Equation (1). Policing must ensure that no contract Q_i exceeds its allocated execution budget. The middleware executes policing code in the context of the server's worker threads. qContracts exports the $contractChargeBudget(client\_id, x)$ primitive for the purpose of policing. The primitive is called upon execution of each service unit, where x is the byte size of the served unit. The call reduces the budget U_iP_i by $max(a + bx,\ cx)$, the bottleneck resource processing attributed to the served unit. If the budget expires the call either blocks or, in its non-blocking version, returns an error. To describe how to achieve policing with this mechanism, we make distinction between two general types of servers:

Servers with long-lived flows: A server is said to have long-lived flows if the single connection's flow lasts much longer than P_i. In that sense, servers with persistent HTTP 1.1 connections, for example, do not necessarily belong to the long-lived flow category since their connections may be idle most of the time with only sporadic bursts of short flows. Video servers, on the other hand, are a good example of this type. Service rate, $R_i = M_i/P_i$, in this case is typically the rate at which the single request is served, e.g., the frame rate at which the movie is transmitted. In servers with long-lived flows, the server's worker thread or the $write()$ library call that sends the frames out to the client is instrumented to call the blocking version of $contractChargeBudget(client\_id, frame\_size)$ upon each frame transmission. The call will block in qContracts when the execution budget expires, and will unblock when it is replenished. Since the budget is replenished to U_iP_i

every period P_i, the average utilization due to request execution on behalf of the i-th client cannot exceed U_i.

Servers with short-lived flows: a server is said to have short-lived flows if a large number of server responses can be sent to completion within P_i. This, for example, is true of web servers which typically have the capacity to process hundreds of requests per second. In servers with short-lived flows, contracts are typically defined on flow aggregates. Service rate, $R_i = M_i/P_i$, in this case, defines the aggregate request rate (e.g., the hit rate on a particular web site). Bandwidth W_i defines the aggregate delivered byte rate. In this case, the use of the blocking version of $contractChargeBudget(client\_id, x)$ is inappropriate for policing because, upon budget expiration, the call blocks only the calling thread. It leaves it possible for a different thread to pick another request whose processing is charged to the same contract. In the worst case, all server threads may encounter, during some period P_i, requests charged to a contract whose budget has expired. All these threads will thus block on $contractChargeBudget(client\_id, x)$, bringing the entire server to a halt until that budget is replenished. To avoid this problem, in servers with short-lived flows, the $read()$ library call is instrumented to call $contractCheckBudget(client\_id)$ as each request is read in. The latter call returns an error if the budget has expired, in which case the instrumented $read()$ will discard this request (for violation of the contracted rate) and read the next. The $write()$ library call that sends the response to the client is instrumented to call a nonblocking $contractChargeBudget(client\_id, response\_size)$ upon response transmission to update the budget accordingly. (Note that if server source is available, it may be easier to call $contractCheckBudget(client\_id)$ and $contractChargeBudget(client\_id, response\_size)$ directly from the server's worker thread.) Since no requests are served after the $U_i P_i$ budget expires, and since the budget is replenished every period P_i, the average utilization by the i-th client cannot exceed U_i.

4.1.4 Profiling

The goal of profiling is to estimate the parameters a, b and c, introduced in Section 4.1.1, that describe the different overhead components per service unit (i.e., a unit of M_i), so that QoS mapping can translate QoS specification correctly into bottleneck resource utilization. During profiling, the middleware estimates overall server utilization, U, by computing the fraction of time the server was busy processing requests. In addition, the middleware computes the aggregate load executed within some interval T by counting the total number of times, say N, that the primitive $contractChargeBudget(client\_id, x)$ was called within that interval (for any contract), and aggregating the sum S of x values with which the primitive was called. The independently-measured utilization U within some period T should be equal to $max(aR+bW, cW)$ where $R = N/T$ and $W = S/T$. The profiling measures U, R and W in successive periods T and uses a regression analysis to refine the estimate of a, b and c online to fit the equation $U = max(aR + bW, cW)$. Profiling stops when a stable estimate is reached. The parameters a, b and c are platform-independent and need to be re-evaluated only when the platform is upgraded.

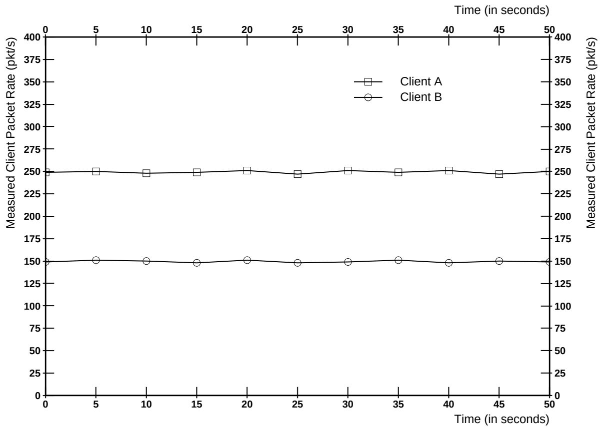


Figure 1. Performance isolation

To assess the goodness of our QoS-provisioning mechanism, we present the results of a sample experiment. We wrote a "dummy server" and created two contracts, Q_A and Q_B, for client classes A and B respectively. The contracted service rate was specified to be 250 pkts/s for Q_A and 150 pkts/s for Q_B. Data was sent to each class via a persistent infinite loop, simulating an overload condition. Data received by each class was collected on a destination machine (for simplicity we sent all data belonging to the same class to the same destination). Figure 1 shows the delivered packet rate of each class, measured at their respective destinations. It can be seen that the rate follows the contracted specification exactly. Although two data streams were generated by instanciations of the same infinite loop, each stream was policed by our middleware to a different rate as specified in its QoS contract. The experiment demonstrates the middleware's success in achieving per-class QoS provisioning and performance isolation. In Section 5 we present a similar experiment using a real web server.

4.2 Performance Differentiation

QoS-guaranteed clients need to receive preferential treatment over non-guaranteed clients in case of overload. In a real-time system, priorities are a commonly-used mechanism for performance discrimination on the basis of importance. However, in contemporary mainstream multi-threaded servers (e.g., web and video servers) all worker threads have the same priority. Performance differentiation requires that requests whose processing is charged to a guaranteed contract be served first. A straightforward approach is to alter server thread priorities in middleware (e.g., in the $read()$ call when a request is read) such that the priority depends on the request being served by the thread. The approach works well when the CPU is the bottleneck resource. Unfortunately, web and multimedia servers also consume memory buffer and network bandwidth, which may become bottlenecks, thus making thread priorities ineffective. In fact, one may argue that due to the extremely low cost of processing power (e.g., PCs) relative to that of dedicated communication links, the latter is likely to be the bottleneck in most server installations. To design a proper performance-differentiation scheme, we first investigate the effects of non-CPU bottlenecks on priority-based scheduling. We constrain ourselves to differentiation between two traffic classes; guaranteed and non-guaranteed. We present a middleware solution for performance differentiation that does not require OS modification. Performance analysis of extensions to multiple priority levels is left for future work.

4.2.1 The Memory Buffer Effect

An internal communication data buffer is associated in the OS with each outgoing connection. When the network is slow, the server causes these communication buffers to saturate with the server's outgoing data awaiting transmission. Figure 2 plots the observed ratio of bandwidth delivered by the server to two clients (or client classes), A and B, versus the buffer size ratio of their respective connections when the aggregate data sent to the clients saturates the server's 10 Mb/s Ethernet link causing an overload. In the experiment, client A has a QoS contract for a packet rate of $R_i = M_i/P_i = 600$ pkts/s (1500 bytes/pkt). Client B is not-guaranteed. The figure shows both (i) the case where the server threads serving the two clients have the same priority and (ii) the case where the thread serving the guaranteed client has higher priority. It can be seen that regardless of thread priority, the delivered bandwidth ratio followed closely the connection buffer-size ratio showing a strong correlation between outgoing buffer size and connection bandwidth. Note that the buffers are in the OS, and, as such, are inaccessible to the middleware. Throughput dependency on buffer size is the first problem we need to overcome in order to allow a higher-priority service thread to take resources from a lower-priority one even when the network is the bottleneck. A solution is described in Section 4.2.3.

4.2.2 The Semaphore Effect

Semaphores are often used by the operating system to block kernel threads, e.g., when accessing common data structures while performing I/O operations. Most non-

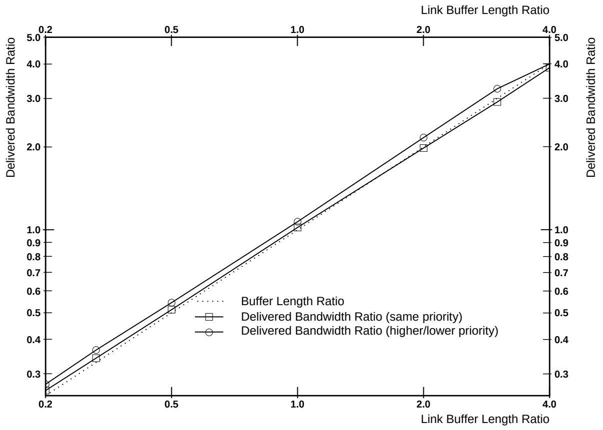


Figure 2. Bandwidth and buffer size

real-time operating systems do not ensure that when a semaphore is signaled, the highest-priority thread waiting on the semaphore gets the CPU. Instead, a waiting thread is often awakened in FIFO or LIFO order. This deficiency poses a difficulty when implementing service differentiation. For example, when the network device driver becomes saturated, threads may be blocked when they attempt to deposit data into its full buffer. These threads may later be unblocked in an order that does not respect their priority.

The solid lines in Figure 3 show the packet rate received from the server by clients A and B on their respective machines when the server generates data for each client at the rate shown by the dotted line. In the experiment, client A is guaranteed while client B is not. It can be seen that each client receives all server data up until the aggregate packet generation rate at the server increases beyond $700pkt/s$ (i.e., until $R_i = 350pkt/s$ for each client). This aggregate rate saturates the 10Mb/s ethernet. The server then fails to deliver packets to both clients even although there is enough bandwidth to serve the higher priority client alone. We conclude that clients are not served in priority order.

In order to verify that the semaphore implementation is the cause of the observed priority-inversion problem, we changed the semaphore implementation such that the highest-priority waiting thread is resumed when a semaphore is signaled. The above experiment was repeated. Figure 4(a) shows that the higher-priority client can now achieve its ideal service rate, R_i, much more closely at the expense of the lower-priority client when the network saturates. Note, however, that we are looking for solutions where the OS internals are not touched. Such solution is described next.

4.2.3 A Middleware Solution to Service Differentiation

The problems mentioned above pose a significant challenge to implementing proper service differentiation without accessing server and OS code. We noticed, however, that

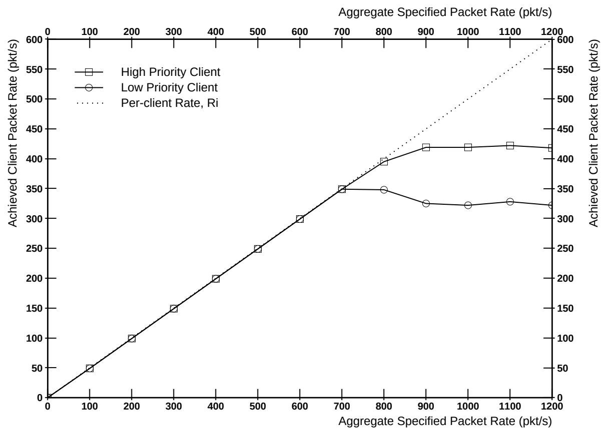
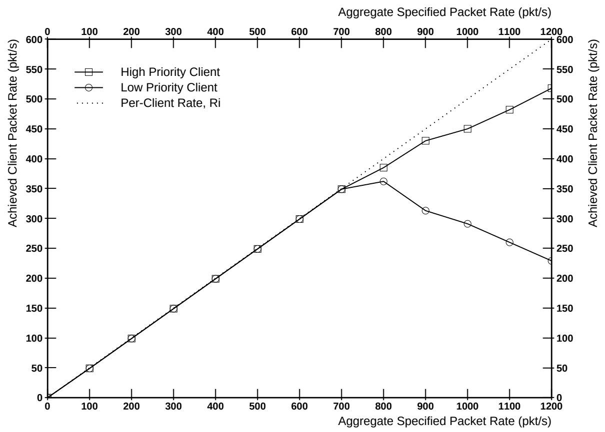


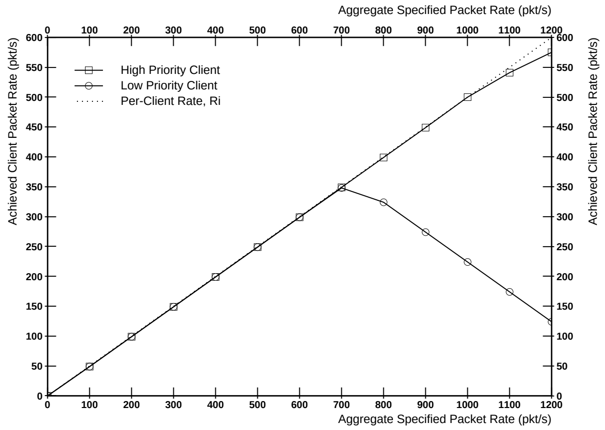
Figure 3. Priority-insensitive semaphores

these problems arise only under overload. In the absence of overload, the system performs to specification, e.g., as seen in Figure 3, for aggregate packet rates below 700 pkts/s. We, therefore, utilized our performance-isolation mechanism to "emulate" priorities. A utilization budget is created for lower-priority clients. Its value is set to the remaining system capacity left unused by guaranteed clients, i.e., to $100\% - \sum_i U_i$. The budget is treated as yet another QoS contract. The contract is policed not to over-draw its allocated resources, as described in Section 4.1. Server thread priorities are not altered, not are OS communication buffers and semaphores. Worker threads execute client requests at the same machine priority. Performance differentiation is achieved by means of policing the non-guaranteed clients to prevent them from over-drawing those resources logically allocated to guaranteed ones.

Figure 4(b) illustrates the goodness of this approach at mimicking the thread prioritization effect. The experiment of Section 4.2.2 was repeated with one guaranteed and one non-guaranteed client. The non-guaranteed connection was now policed to prevent overload. When the network is saturated, the data rate of the non-guaranteed client is forced to decline, via policing, with the increase in the contracted rate of the guaranteed client to keep their sum constant at the maximum network bandwidth. It can be seen from Figure 4(b) that the guaranteed client is now able to receive its contracted service rate. By adding the received packet rates of the two clients in Figure 4(b) we can see that the maximum achieved rate is about 720 pkts/s. The corresponding maximum network bandwidth is approximately 8.64 Mb/s (we send 1500 byte packets on a 10 Mb/s Ethernet). The results shown in this subsection demonstrate that performance differentiation can be achieved with the middleware without modifying actual thread priorities, semaphore implementation, and communication buffer size. Together, our implementation of performance isolation and prioritization allows for augmenting the original server software



(a) Fixing the semaphore problem



(b) Prioritization via policing

Figure 4. Performance differentiation

with QoS-provisioning and performance-differentiation capabilities. An example with a real server is presented in the following section.

5 Web Server Application

We now describe an application of the proposed middleware. In particular, we show how to endow an Apache web server with QoS-guaranteed behavior. Apache implements the server side of the HTTP protocol to retrieve and manipulate web content upon client requests. Ordinarily, no subscription to the service is required. For better concurrency, the server implements multiple worker threads (e.g., on a Windows NT platform) or processes (e.g., on a UNIX platform) which together listen on a common TCP socket for client requests. All requests are queued FIFO in the common socket queue. Once a request is dequeued by some thread or process, a TCP connection is established with the corresponding client, and the request is handled.

We compiled an Apache 1.3.3 server on an Ultra Spark workstation running Solaris. The workstation was connected via a 10Mb/s ethernet to three other machines which together simulated the end-user community. We used the web load generation tool, httperf [22], on each of these three machines to generate load on the server. The tool generated a sequence of requests for 10KB URLs. The maximum achieved request rate was about 100 reqs/s, making a data throughput of about 8Mb/s. We created two sites on the Apache server. A QoS contract was established with one of the two sites via qContracts. The contract specified a guaranteed maximum service rate R_i of 30 URLs/second with a $10KB$ average URL size. The other site was not guaranteed. The QoS contract was stored in a configuration file accessed by the middleware. The Apache server already parses incoming HTTP requests, and determines the accessed site specified in the request header. We therefore call $contractsCheckBudget()$ when Apache's $ap\_read\_request()$ returns, passing it the site ID. If $contractsCheckBudget()$ returns an error indicating that the site's budget has expired we discard the request by closing its TCP connection (using Apache's $ap\_bclose()$). Similarly, when Apache's $ap\_process\_request()$ function returns indicating that a reply was sent, we call a non-blocking $contractsChargeBudget()$ with the length of the served URL.

Figure 5 demonstrates both QoS provisioning and performance differentiation thus achieved. In this experiment, the number of requests for the guaranteed site was kept constant at 25 reqs/s, while that for the non-guaranteed site was increased, causing the server to be overloaded. Once the system was overloaded, an increasing fraction of non-guaranteed connections were rejected. Figure 5 shows the rate at which requests are served for both the guaranteed and non-guaranteed sites. It can be seen that while all requests for the guaranteed site are served successfully, the other site's service rate declines. The decline (as opposed to saturation) in service rate seen by the non-guaranteed site is due to the increase in resource consumption wasted on rejected connections which reduces the remaining server capacity and throughput of the machine. Note, from Figure 5 that (i) the guaranteed site receives its contracted QoS, and (ii) the non-guaranteed site is served at a "lower priority" under overload so that it does not infringe on the resources assigned to the guaranteed site. Thus, the middleware is successful in achieving its goals.

The maximum throughput of the server was compared to that without the middleware present. The difference was in the range of the measurement noise, which was less than 1%. The low overhead of the middleware is attributed to several factors. First, it executes in user space in the context of server threads. Thus, qContracts calls are local function calls in the same address space. To regulate aggregated flows, the middleware relies totally on monitoring and admission control. Monitoring overhead is that of manipulating the rate and bandwidth counters when $write()$ is performed. Admission-control overhead is that of checking the counter values. Based on these values, admission control either does nothing (the default), or closes the present TCP connection (to reject a request). Its overhead on admitted requests is therefore minimal. In our tests, request classification is already performed by Apache in the standard $ap\_read\_request()$ call which determines the accessed URL. Classification overhead is therefore the same whether qContracts are present or not. The middleware imposes no overhead components other than those mentioned above, which explains its low cost.

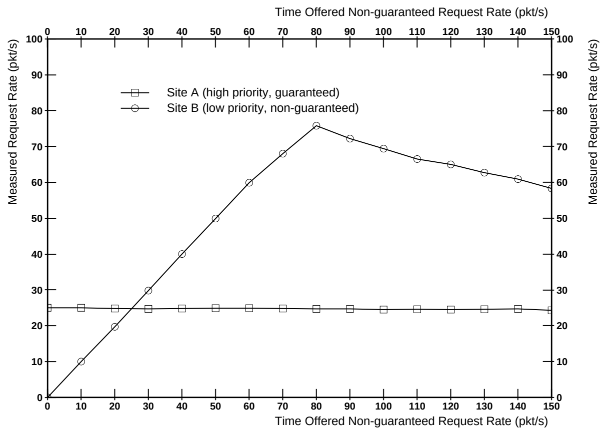


Figure 5. QoS provisioning

6 Discussion

We have developed and evaluated new middleware for future performance-assured services. The middleware introduces programming abstractions and mechanisms that realize soft QoS guarantees. It employs transparent self-profiling mechanisms to decouple application code from assumptions on the underlying platform speed which improves portability of QoS-sensitive applications. Not surprisingly, our experiences with this middleware indicate existence of a significant difference in the means of enforcing QoS contracts depending on flow length. For example, multimedia applications are dominated by long-lasting connections of considerable bandwidth delivered to the client. QoS enforcement is best achieved by policing individual outbound server flows. A bursty flow-generation process may be smoothed by blocking the generating server thread when it exceeds its maximum contracted rate. This approach does not work for controlling aggregated short-lived flows such responses of web servers. As shown in Section 4.1.3, policing aggregated outbound flows from a site by blocking the sending server threads may cause the entire server to halt unless different thread pools serve different sites. Assigning different server thread pools to different traffic categories is therefore essential for proper QoS management of aggregate flows whenever the underlying OS mechanisms or network traffic shaping mechanisms impose upper bounds on aggregated flow bandwidth. Examples of such mechanisms include proportional share OS resource allocation, and weighted fair queuing.

To reuse today's best-effort server code, where a single pool of threads or processes serves all requests, our solution is to resort to request admission control to make the outgoing aggregate flow bandwidth conform to specifications. Since the resource requirements imposed by a single service request are often unknown, *measurement-based* admission control is needed to regulate the fraction of admitted requests based on bandwidth measurements. In web servers, for example, the length of requested URLs can differ by a couple of orders of magnitude from one request to another. Measurement-based admission control will be effective as long as the resource requirements of the largest request are a small fraction of total server capacity, making it possible to apply the laws of large numbers to aggregated flows and use a fluid flow model.

An important question with admission control is which requests to reject. One might argue that indiscriminate rejection is acceptable within the same traffic class. This, however, is not always a good policy. If clients' sessions are comprised of multiple requests, all clients are likely to see rejections during the session, unless admission control is adjusted such that a subset of clients are rejected consistently under overload, while others see consistently good service (i.e., actually complete their sessions). In [1] such an admission control mechanism is described to ensure QoS consistency to served clients.

Both policing and measurement-based admission control can be used to implement service differentiation. A low-priority aggregate flow is forced (by either method) to fit within the bandwidth left unused by higher-priority flows. This mechanism successfully "fakes" priorities in middleware, even when identical same-priority threads handle different traffic classes in the server. It requires an estimate of aggregate server capacity as well as a measurement of the bandwidth used by each priority class. While the approach works well for a limited number of classes, it is unclear if can scale up to a large number of priority levels because admission control in each level depends on measurements of all higher priority levels, and measurement accuracy decreases with the granularity of a traffic class.

Another important problem that arises with flow prioritization is that of consistent flow-priority management across multiple resources. Unless the CPU is the bottleneck resource, threads will block waiting on the real bottleneck which must then be allocated in the same priority-sensitive fashion. An example of priority-insensitive behavior when the bottleneck resource (a memory buffer) isn't allocated properly is shown in Section 4.2.1.

Finally, classification of requests into one of several flows poses a particularly important concern. Such classification is often dependent on application-specific data (e.g., the site named in the HTTP header). Implementing request classification in the socket library transparently to the server may result in a large run-time overhead since it will require parsing all requests and interpreting their content (such as the HTTP header fields) in the context of $read()$ calls. Classification, however, is already performed in the context of usual server execution, so it is advantageous to have access to server code. Unfortunately, performing classification in user space (i.e., in the server or middleware) is not efficient for another reason. Since admission control is performed *after* classification (to discriminate among different classes), rejected requests will have consumed a substantial amount of resources in the kernel by the time they are rejected by the server. A substantial amount of execution resources can thus be wasted on eventually rejected connections. This concern calls for OS support for early classification in the kernel. Such OS mechanisms are outside the scope of this paper.

7 Conclusions

We presented a new resource-management mechanism for web and multimedia servers to achieve service rate and bandwidth guarantees. The mechanism is embedded in a middleware layer interposed between a standard best-effort server and a non-real-time OS. It provides support for managing QoS contracts that implements QoS provisioning and performance differentiation. QoS contracts are specified in platform-independent QoS parameters, leaving it to the underlying system to translate these parameters into host-dependent resource requirements and account for those requirements to achieve performance isolation and differentiation. The use of the middleware requires no architectural modifications to existing server design and request handling mechanisms, and imposes no special-purpose requirements on the operating system. The middleware was shown to be effective in achieving the desired QoS for soft real-time applications.

References

[1] T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[2] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[3] T. F. Abdelzaher and K. G. Shin. End-host architecture for qos-adaptive communication. In *IEEE Real-Time Technol-*

ogy and Applications Symposium, Denver, Colorado, June 1998.

[4] C. Aurrecoechea, A. Cambell, and L. Hauw. A survey of QoS architectures. In *4th IFIP International Conference on Quality of Service*, Paris, France, March 1996.

[5] S. Brandt and G. Nutt. A dynamic quality of service middleware agent for mediating application resource usage. In *Real-Time Systems Symposium*, pages 307–317, Madrid, Spain, December 1998.

[6] A. Cambell, G. Coulson, and D. Hutchison. A quality of service architecture. *ACM Computer Communications Review*, April 1994.

[7] D. Chen, R. Colwell, H. Gelman, P. K. Chrysanthis, and D. Mosse. A framework for experimenting with QoS for multimedia services. In *International Conference on Multimedia Computing and Networking*, 1996.

[8] B. Field, T. Znati, and D. Mosse. V-net: A framework for a versatile network architecture to support real-time communication performance guarantees. In *InfoComm*, 1995.

[9] R. Fielding and G. Kaiser. The apache HTTP server project. *IEEE Internet Computing*, pages 88–90, July 1997.

[10] R. Gopalakrishnan and G. Parulkar. Efficient user space protocol implementations with qos guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 1998.

[11] P. Goyal, X. Guo, and H. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of Second Usenix Symposium on Operating System Design and Implementation*, Seattle, Washington, October 1996.

[12] M. Humphrey, S. Brandt, G. Nutt, and T. Berk. The DQM architecure: middleware for application-centered QoS resource management. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, San Francisco, California, December 1997.

[13] K. Jeffay, D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Real-time Systems Symposium*, pages 480–491, Madrid, Spain, December 1998.

[14] M. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[15] M. B. Jones and P. J. Leach. Modular real-time resource management in the rialto operating system. Technical Report MSR-TR-95-16, Microsoft Research, Advanced Technology Division, May 1995.

[16] A. Lazar, S. Bhonsle, and K. Lim. A binding architecture for multimedia networks. *Journal of Parallel and Distributed Computing*, 30:204–216, Nevember 1995.

[17] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in real-time mach. In *Proceedings of Multimedia*, Japan, March 1996.

[18] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach". In *Proceedings of the Real-time Technology and Applications Symposium*, June 1996.

[19] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *JSAC*, June 1997.

[20] A. Mehra, A. Indiresan, and K. G. Shin. Structuring communication for quality of service guarantees. In *IEEE Real-Time Systems Symposium*, pages 144–154, Washington, DC, December 1996.

[21] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[22] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[23] K. Nahrstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[24] K. Nahrstedt and J. Smith. Design, imlementation, and experiences with the OMEGA end-point architecture. *IEEE JSAC*, September 1996.

[25] J. Nieh and M. S. Lam. The design, implementation, and evaluation of SMART: A scheduler for multimedia applications. In *16th ACM Symposium on Operating System Principles*, pages 184–197, Saint-Malo, France, October 1997.

[26] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[27] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for qos-based resource allocation problems. In *Real-time Systems Symposium*, pages 296–306, Madrid, Spain, December 1998.

[28] D. Rosu, K. Schwan, and S. Yalamanchili. FARA - a framework for adaptive resource allocation in complex real-time systems. In *Real-time Technology and Applications Symposium*, pages 79–84, Denver, Colorado, June 1998.

[29] S. Sommer and J. Potter. Operating system extensions for dynamic real-time applications. In *IEEE Real-Time Systems Symposium*, pages 45–50, Washington, DC, December 1996.

[30] B. Srinivasan, S. Pather, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Real-time Technology and Applications Symposium*, pages 112–120, Denver, Colorado, June 1998.

[31] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional-share resource allocation algorithm for real-time time-shared systems. In *Real-Time Systems Symposium*, pages 288–299, Washington, DC, December 1996.

[32] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop*, pages 73–82, October 1990.

[33] C. Volg, L. Wolf, R. Herrwich, and H. Wittig. HeiRAT – quality of service management for distibuted multimedia systems. *Multimedia Systems Journal*, 1996.

[34] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[35] D. K. Y. Yau and S. Lam. Migrating sockets for networking with quality of service guarantees. In *International Conference on Network Protocols*, Atlanta, Georgia, October 1997.