User-Level QoS-Adaptive Resource Management in Server End-Systems

Tarek F. Abdelzaher, *Member*, *IEEE*, Kang G. Shin, *Fellow*, *IEEE*, and Nina Bhatti, *Member*, *IEEE*

Abstract—Proliferation of QoS-sensitive client-server Internet applications such as high-quality audio, video-on-demand, e-commerce, and commercial web hosting has generated an impetus to provide performance guarantees. These applications require a guaranteed minimum amount of resources to operate acceptably to the users, thus calling for QoS-provisioning mechanisms. One good place to locate such mechanisms is in server communication subsystems. Server-side communication subsystems manage an increasing number of connection end-points, thus readily controlling important bottleneck resources. We propose, implement, and evaluate a novel communication server architecture that maximizes the aggregate utility of QoS-sensitive connections for a community of clients even in the case of overload. A contribution of this architecture is that it manages QoS from the user space and is transparent to the application. It does not require modifications to the OS kernel, which improves portability and reduces development cost. Results from an experimental evaluation on a microkernel indicate that it achieves end-system overload protection and traffic prioritization, improves insulation between independent clients, adapts to offered load, and enhances aggregate service utility.

Index Terms—Internet servers, operating systems, QoS, resource management.

1 Introduction

QoS-sensitive resource management mechanisms on server end-systems are motivated by the multitude of emerging Internet applications, such as multimedia streaming and e-commerce, which require predictable performance and contractual performance guarantees. To address this issue, this paper describes a novel communication server architecture for QoS-adaptive resource management which exports the abstraction of "QoS contracts" between the server and clients. The architecture augments current web and multimedia services with QoS enforcement mechanisms, which achieve overload protection, ensure performance isolation between independent connections or classes of connections on the server end-system, implement connection prioritization across multiple resources, and perform graceful QoS-adaptation to dynamically changing load conditions.

QoS contract enforcement in our architecture lies transparently beneath the application by exploiting dynamic shared libraries to provide legacy applications with QoS extensions without modifying application code. The libraries communicate with a separate QoS-aware communication server process on top of an operating system kernel that supports threads and fixed priority scheduling. Our experiments demonstrate the usefulness and efficacy of the architecture in terms of achievement of QoS guarantees. A key contribution of the architecture is the transparent implementation of QoS enforcement mechanisms in user space. We also implement policies for QoS optimization that maximize utility under resource constraints. The architecture is evaluated on a microkernel operating system.

The rest of this paper is organized as follows: Section 2 elaborates on the notion of QoS used in this paper. It proposes a flexible form of QoS contracts suitable for emerging QoS-sensitive services. Section 3 describes our architecture for embedding QoS provisioning into best-effort server platforms. Section 4 describes utility-optimizing resource allocation policies. Section 5 presents and evaluates mechanisms for transparently *enforcing* resource allocation and achieve performance guarantees in the absence of kernel support. Section 6 describes related work. Finally, the paper concludes with Section 7.

2 The QoS Contract

In a QoS-aware service, QoS requirements must be specified to the server's communication subsystem. In our architecture, this specification is expressed in a *QoS contract*. To express the flexibility of adaptive applications, our QoS contract model assumes that the service exports multiple QoS levels of different quality and utility to the user. A QoS contract C_i with client i contains 1) a desired QoS level, $L_{desired_i}$, 2) the minimum QoS level acceptable to that user $L_{min_i} \leq L_{desired_i}$, and 3) the utility (or charge rate) $R_i[k]$ for each QoS level L_k exported by the service in the range $[L_{min_i}, L_{desired_i}]$. Note that the charge rate, $R_i[k]$, for the same QoS level depends on the client, i. For example, the service may support individual rates, corporate rates, promotional rates, and frequent buyer rates. A QoS-violation penalty, V_i, may be defined for failing to meet the requirements of the minimum level L_{min_i} of an established contract. It is useful to think of contracts as having an extra QoS level, called the *rejection level*, with no resource requirements (no service) and a "reward" of $R_i[k] = -V_i$. It quantifies the penalty of disrupting service to the client (e.g., closing the connection in the middle of transmitting a movie). This model was first proposed by the authors in [4].

The interpretation of QoS levels and the nature of clients who sign the contract with the service depend on the application. In applications such as video-on-demand, where clients request an online movie transmission, QoS levels may represent frame rates and average frame sizes. The contract is signed between the server and the requesting user. In other applications, such as commercial web hosting, QoS levels may specify the server capacity allocated to a hosted site. The contract C_i is signed with the content provider (i.e., the hosted web site's owner). In both cases, it suffices to specify a QoS level k of contract C_i with the following two parameters:

- *Aggregate Service Rate*, $\mu_i[k]$: expressed as $M_i[k]$ *units of service* per specified period $P_i[k]$, i.e., $\mu_i[k] = M_i[k]/P_i[k]$. The units of service are arbitrary, but all contracts with a particular server must use the same unit. Examples of service rate are: $M_i[k]$ served URLs per period (e.g., in web servers), $M_i[k]$ served packets per period (e.g., in communication subsystems), or $M_i[k]$ served frames per period (e.g., in audio/video servers).
- *Aggregate Data Bandwidth*, $W_i[k]$: specifies the aggregate bandwidth in bytes per second to be allocated for the contract. Aggregate bandwidth is orthogonal to service rate because the unit of service (such as a request, frame, or packet) does not necessarily have a fixed number of bytes.

Aggregate service rate and data bandwidth are useful QoS parameters because resource consumption at the end-system can be approximated by two components: 1) a fixed average

- *T.F. Abdelzaher is with the Department of Computer Science, School of Engineering and Applied Science, University of Virginia, 151 Engineer's Way, PO Box 400740, Charlottesville, VA 22904-4740. E-mail: zaher@cs.virginia.edu.*
- *K.G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122. E-mail: kgshin@cs.umich.edu.*
- *N. Bhatti is with Hewlett Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304-1126.*

Manuscript received 19 Jan. 2000; revised 26 July 2002; accepted 31 July 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111244.

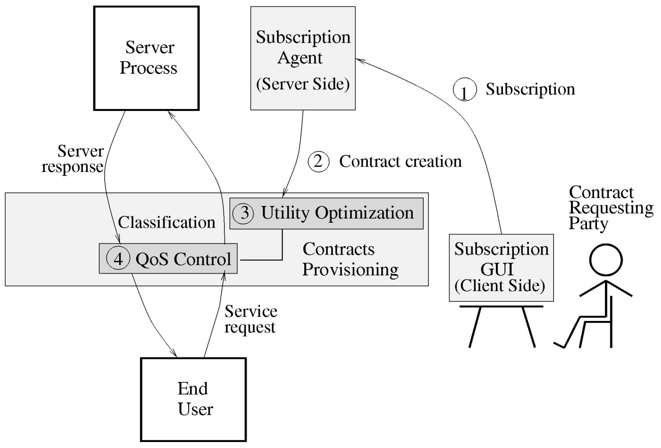


Fig. 1. Architecture for QoS.

per-unit-of-service consumption (such as per-packet protocol-processing cost) and 2) a data-size-dependent consumption (such as data copying and transmission cost). This approximation becomes increasingly valid with increased levels of aggregation. We do not deal with jitter and end-to-end response-time constraints since their satisfaction depends largely on network support that cannot be guaranteed by the server end-system alone.

3 General Architecture

Fig. 1 gives a high-level view of our architecture for performance-assured services, showing important components and their interactions. The shaded regions are the software components we add to the existing infrastructure to provide QoS-contract guarantees. In our model, customers desiring QoS provisioning (e.g., the owners of a web site to be hosted by the service) will subscribe to receive "guaranteed" service. Subscriptions are processed via a subscription agent, which is a process or CGI script separate from the server process, invoked on the server machine. The agent creates QoS contracts with the machine's communication subsystem on behalf of the subscribed customer by calling the QoS-sensitive API extensions exported by our communication subsystem. Contracts are admitted if enough resources exist for their execution. Contract admission guarantees that the customer will receive service at one of the acceptable QoS levels specified in the contract or be paid the QoS violation penalty. The server reserves the right to change the QoS level dynamically. A utility-maximizing policy, invoked at contract admission time, (re)computes the "right" QoS level for each contract given the current resource availability and demand.

QoS levels are enforced by *QoS control*, which essentially maintains the resource allocation necessary to meet QoS level requirements. Since we are interested in portable QoS control mechanisms, resource allocation is achieved without OS kernel enforcement. Our approach is to rely on policing mechanisms to ensure that actual resource consumption coincides with the logical resource allocation. In services with long-lived flows and per-flow contracts (e.g., video transmission), load is controlled most efficiently via flow-control mechanisms that police the outbound server connections to limit resource consumption, so it does not exceed its allocation. In services with short-lived flows and contracts defined on flow aggregates (e.g., web hosting), load is controlled more efficiently by policing the inbound request rate via admission control. Request admission control is *not* to be confused with admission of new contracts into the system. While the latter mechanism ensures that the machine is not overloaded as a whole, the former ensures that each individual contract does not use more resources than its allocated share.

4 Utility Optimization

The key goal of our QoS architecture is to optimize global utility across the community of clients. Such optimization is achieved by proper resource allocation. The QoS optimization policy is determined in a replaceable *policy module*. The module is a self-contained function that accepts as input a data structure containing the currently accepted contracts as well as any contracts considered for admission. The output of the module is the QoS level chosen for each contract. The module also uses the output of the profiling subsystem to determine execution overheads, as will be described in Section 4.1. Quantification of these overheads is essential for server capacity planning. The output of the policy module (i.e., the selected QoS levels of each admitted contract) is enforced by a separate mechanism that is independent of how policy decisions were made. In the subsequent sections, we describe the three main components of the architecture, namely, the profiling subsystem, the policy module, and the enforcement mechanism, respectively.

4.1 The Profiling Subsystem

The resource requirements imposed by each QoS contract must be known before utility optimization can be made. In [2], we reported, in the context of web servers, that the consumption of resource j on the end-system due to processing of a unit of service (e.g., packet, frame, or URL) is accurately approximated by $a_j + b_j x$ (where a_j and b_j are constants that depend on the consumed resource and x is the size of data served). Parameters a_j, b_j are determined by profiling. In [1], we report on our experimentation with online estimation of these parameters using resource monitoring and regression analysis applied to CPU and network resources. Stable and accurate parameters estimates are obtained. They need to be reevaluated only when the platform is upgraded. The sensitivity of these parameters to load variations is found small enough to make it possible to use their worst-case values for admission control without underutilizing the system.

Let each created contract C_i have multiple acceptable QoS levels such that the requirements of QoS level k for resource j are given by $U_i^j[k]$ and the utility of delivering this QoS level is $R_i[k]$. Aggregating the capacity consumed by processing a sequence of service units during some observation period, the resource utilization required to meet the requirements of QoS level, k, of contract C_i, is given by:

$$U_i^j[k] = a_j M_i[k]/P_i[k] + b_j W_i[k], \tag{4.1}$$

where $M_i[k]/P_i[k]$ $(= \mu_i[k])$ and $W_i[k]$ are the service rate and bandwidth parameters of the QoS level.

4.2 The Policy Module

In general, to achieve the best resource allocation, all resources such as CPU, disk bandwidth, and communication bandwidth must be considered. In practice, since our QoS guarantees have *throughput* semantics, only the bottleneck resource consumption is relevant. For example, if the bottleneck in some system is disk I/O, system throughput will be determined by the maximum sustainable I/O rate; CPU consumption will be irrelevant. Below, we describe an optimal single resource QoS-level assignment policy and establish the near-optimality of a simple first-come first-served QoS-level assignment. Both policies are implemented in our architecture as replaceable policy modules.

4.2.1 Optimal QoS-Level Assignment

Suppose there are n QoS contracts to be handled by the server. Let contract C_i specify m_i acceptable QoS levels with utility $R_i[1], \ldots, R_i[m_i]$, respectively, and let the QoS-violation penalty be V_i. Each QoS level k is now given by the resource utilization requirements $U_i[k]$ computed as described in Section 4.1 (we omit the resource index here since we consider a single bottleneck resource). We introduce an additional artificial QoS level for each contract, called the *rejection level*, at which the client receives no service. This level has no resource requirements (i.e., $U_i[k] = 0$) and incurs a negative utility equal to its QoS-violation penalty (i.e., $R_i[k] = -V_i$) for an established contract, and 0 for a contract being considered for admission. To reduce the NP-complete optimization problem to a polynomial-time problem, we consider a subclass where $U_i[k]$ can only take discrete values which are multiples of some small constant δ. This problem is solvable in polynomial time with dynamic programming. The optimality of the solution follows directly from the optimality of dynamic programming itself. Dynamic programming produces an optimal solution to a problem if it exhibits two properties; optimal substructure and recursion [8]. Such a formulation is described next.

Optimal substructure. This property means that if the optimal path from A to B passes via C, then the subpath from C to B must also be optimal. We construct a grid of subproblems that satisfy the aforementioned property. Each subproblem $S(i, U)$ is that of selecting an optimal QoS level k_l for the first i contracts, i.e., for $C_l \in \{C_1, \ldots, C_i\}$ such that 1) their total utility $\sum_{1 \le l \le i} R_l[k_l]$ is maximized and 2) the utilization does not exceed U. In the following, we prove by contradiction that this formulation exhibits optimal substructure.

Proof. In the optimal solution for $S(i, U)$, let the first j contracts $C_1, \ldots, C_j$ (where $j < i$) consume U^j resources, and their total utility be W. Assume by contradiction that, due to lack of optimal substructure, W is *not* the optimal solution for $S(j, U^j)$. In this case, by optimally reassigning QoS levels to the first j contracts, their utility can be improved for the same utilization. Hence, the utility of the entire set of i contracts is improved, which contradicts the optimality of $S(i, U)$. This proves by contradiction that our formulation exhibits optimal substructure.

For notational simplicity, we let $S(i, U)$ also denote the resulting aggregate service utility. Given n contracts, we need to solve the problem $S(n, 100)$ of assigning optimal QoS levels to all the n contracts, given the full (100 percent) server capacity.

Recursion. The following recursive relation holds true of the subproblems defined above:

$$S(i, U) = \max_{1 \le k \le m_i} \{R_i[k] + S(i-1, U - U_i[k], \ldots, U - U_i[k])\}. \quad (4.2)$$

For the special case of $i = 1$,

$$S(1, U) = \max_{1 \le k \le m_1} \{R_1[k] | U_1[k] \le U\}. \quad (4.3)$$

This recursive relation is the foundation of our dynamic programming formulation. Since the utilization is discretized, there is only a finite number, $K = (100/\delta)$, of possible utilization values in the range $[0, 100]$. Thus, there are a total of nK subproblems to be solved. Solving all of these problems will take $O(K \sum_{1 \le i \le n} m_i)$, which is equivalent to $O(KnL_{av})$, where $L_{av} = (\sum_{1 \le i \le n} m_i)/n$ is the average number of acceptable QoS-levels per contract. Since K is a constant (albeit potentially large), the complexity of the algorithm is $O(nL_{av})$, i.e., the algorithm is linear in the number of contracts. The complete algorithm is given below.

Algorithm A

```
1 for $i = 1$ to $n$
2     for $U = 0$ to $100$ in steps of $\delta$
3         compute $S(i, U)$ from (4.2)
4 return $S(n, 100)$
```

We shall use this algorithm as a basis for comparison with a simpler QoS-maximizing heuristic to assess the quality of the heuristic solution. This comparison gives insight into mechanisms for QoS optimization for a particular application.

4.2.2 FCFS Assignment

While the above optimal algorithm executes in polynomial time, in practice, it may be preferable to serve clients in a first-come first-served manner such that, once a QoS level is chosen for a client, it is not altered by subsequent arrivals. The policy eliminates unwanted QoS fluctuations during the client's session. It has lower computational complexity and, therefore, lower practical overhead. In this section, we prove analytically that this policy is near-optimal. Assume that the server exports n QoS levels, $L_1, \ldots, L_n$, and has a single bottleneck resource. Assume that the resource requirements of each QoS level are fixed and determined only by the level itself (e.g., the resources needed for movie transmission depend only on image quality and not the identity of the recipient). Let the QoS level with the highest absolute reward be denoted L_{hi} and the QoS level with the highest reward per unit of consumed utilization be denoted by L_{lo}. Let the consumed bottleneck resource utilization $U_i^1[k]$ be denoted by h and l for the two QoS levels, respectively. Assume that contracts assign a random uniformly distributed utility $R_i[k]$ to each QoS level such that the utility of level L_{lo} ranges between Min_{lo} and Max_{lo} and the utility of level L_{hi} ranges between Min_{hi} and Max_{hi}.

The optimal policy will always keep the clients with the largest utility for the same resource consumption. Thus, the optimal policy will achieve, at best, a utility of $max(Max_{hi}/h,\ Max_{lo}/l)$ per unit of resources. Note that if there are enough enqueued clients to always fully utilize the system, $max(Max_{hi}/h,\ Max_{lo}/l) = Max_{lo}/l$. In contrast, if the system is underloaded, current clients can be served at their maximum QoS level and $max(Max_{hi}/h,\ Max_{lo}/l) = Max_{hi}/h$ (since degrading the current clients will only reduce their utility without letting more clients into the system).

The FCFS policy will reserve resources for arriving clients in their arrival order until the processing capacity saturates. Since utility is uniformly distributed, FCFS will achieve the expected utility of $(Max_{hi} + Min_{hi})/2h$ per resource unit if it assigns QoS level L_{hi} and $(Max_{lo} + Min_{lo})/2l$ per resource unit if it assigns QoS level L_{lo}. In general, by assigning L_{hi} to new clients under low load and assigning L_{lo} under high load, FCFS allocation policy can achieve an average utility of $max((Max_{hi} + Min_{hi})/2h,\ (Max_{lo} + Min_{lo})/2l)$ per unit of resource consumption. For the sake of finding a lower bound on achieved utility, the above expression is minimized by setting Min_{hi} and Min_{lo} to zero. In this case, the FCFS achieves half of the optimal utility, which constitutes the lower bound. FCFS is thus proven to be a near-optimal policy.

The difference between the optimal policy and FCFS decreases when the QoS-violation penalty is taken into account. QoS violation penalty is never incurred by FCFS since it never reallocates resources assigned to already admitted clients. The optimal policy can take resources away from initially accepted clients and allocate them to more important ones at the cost of paying the QoS violation penalty. Naturally, the larger the penalty, the less beneficial such resource reassignment may be, and the closer the optimal policy becomes to FCFS.

Fig. 2 depicts simulation results that compare FCFS allocation and fair allocation to an optimal QoS maximizing resource allocation policy. By fair, we mean the prevailing policy in contemporary servers, where each client gets an equal share of resources on the average. All contracts were assumed to have two

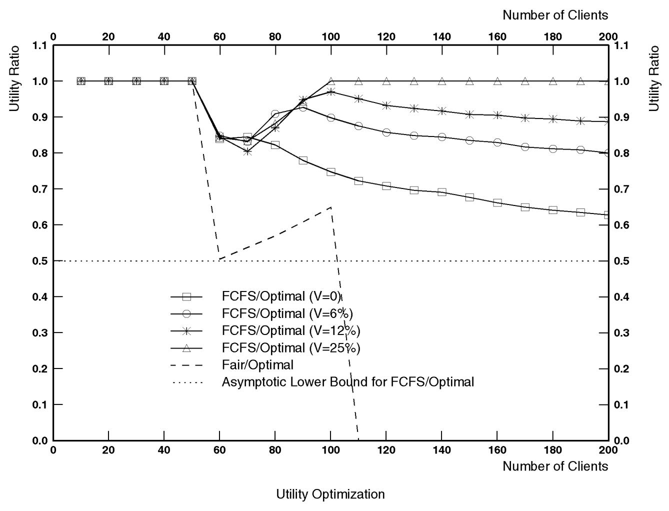


Fig. 2. Near optimal resource allocation.

QoS levels; L_{hi}, which requires 2 percent utilization per client, and L_{lo}, which requires 1 percent. Rewards are uniformly distributed in their respective ranges. The figure plots *average normalized utility*, defined as the aggregate utility achieved for the community of clients by the given resource allocation policy normalized by that of the optimal policy and averaged over 100 experiments. The average normalized utility is plotted versus server load, expressed in the number of accessing clients. Note that the maximum number of clients supportable at QoS level L_{hi} is 50 and the maximum number supportable at L_{lo} is 100. Thus, the server is underutilized when clients < 50, and overloaded when clients > 100. The FCFS policy assigns the highest QoS level, L_{hi}, at low load. At high load, it assigns the QoS level with the highest reward per unit of resource consumption. Several curves are shown for the FCFS policy which differ in the QoS violation penalty V of the application, expressed as percentage of maximum reward.

The figure shows that FCFS is trivially optimal (by selecting QoS level L_{hi}) when the server is underutilized. As load increases, the performance of FCFS drops since assigning L_{hi} may waste scarce resources. Eventually, as load increases, our FCFS policy switches to assigning L_{lo} to incoming clients, thus approaching the optimal policy again. When it becomes impossible to serve all clients, the optimal policy, unlike FCFS, can increase utility further by replacing current less important clients by arriving more important ones, assuming the QoS violation penalty is small. As more clients access the server, the efficacy of such replacement increases, thus increasing the optimal aggregate utility over that achievable by FCFS. This explains the slight decline in the relative performance of FCFS as the load increases beyond 100 clients in Fig. 2. It also explains why FCFS is closer to the optimal when the QoS violation penalty is higher. For critical applications (such as e-commerce) where the QoS violation penalty is very high, FCFS becomes optimal for a large range of load conditions.

The figure also shows that fair resource distribution quickly approaches zero utility in a staircase fashion as the machine gets overloaded, thus motivating QoS-sensitive resource allocation. A drop in utility is seen with fair distribution when per-client resource allocation decreases below the minimum requirements of a particular QoS level.

5 QoS Control

We describe our implementation of a multithreaded communication server with QoS extensions for expressing and enforcing QoS contracts and their resource allocation. The communication server has been implemented on the Open Group microkernel Mk7.2. It exports a socket API to the application, with limited QoS extensions for use by the subscription agent.

5.1 A User-Space QoS Enforcement Architecture

Fig. 3 depicts our QoS enforcement architecture. In order to achieve per-contract QoS, we use a thread-per-contract model in which each contract is allocated one outbound contract handler thread for outgoing traffic and one inbound contract handler thread for incoming traffic. The pool of all contract-handler threads constitutes the server's worker threads that process contracted connections. For connections with no QoS contracts, a default handler pair is used. This pair is created at server boot time and assigned a lower priority than that of other handlers, all of which are assigned the same priority in the kernel. A user-level scheduler implemented in the communication server is responsible for sequencing the contract-handler threads, thus decoupling their QoS-specific scheduling policy from the generic fixed-priority scheduling support in the kernel. Sequencing of contract-handler threads is implemented by associating a semaphore with each. Each thread calls the user-level scheduler after transmission of each packet. The scheduler determines which thread to run next, signals its semaphore, and blocks the caller on the caller's semaphore. If the network link is the bottleneck, a complementary mechanism is needed to prioritize traffic transmission on the link. For this purpose, outgoing packets are queued in a common heap within the server sorted by handler priority and are dequeued in priority order when the network device is not busy.

Contract handlers are implemented as *quasi-periodic* threads. The abstraction of quasi-periodic threads is novel and is a contribution of this work. A quasi-periodic thread Q_i serving contract C_i is a thread that can be executed periodically for a finite duration, starting at an arbitrary point in time. It has a period and budget (per resource) that can, in general, change dynamically, depending on the current QoS level assigned to the contract (recall from Section 2 that each QoS level k of contract C_i specifies a period $P_i[k]$, a budget $M_i[k]$, expressed in units of service, and a data bandwidth $W_i[k]$). Fig. 4a depicts an instance of executing a quasi-periodic thread with a single resource budget.

For each quasi-periodic thread, Q_i, the communication subsystem scheduler maintains a thread data structure that associates Q_i with an underlying fixed-priority kernel thread T_i, an underlying kernel semaphore S_i, a period P_i, a periodic budget vector E_i (in which each element E_i^j gives the budget of the jth resource),

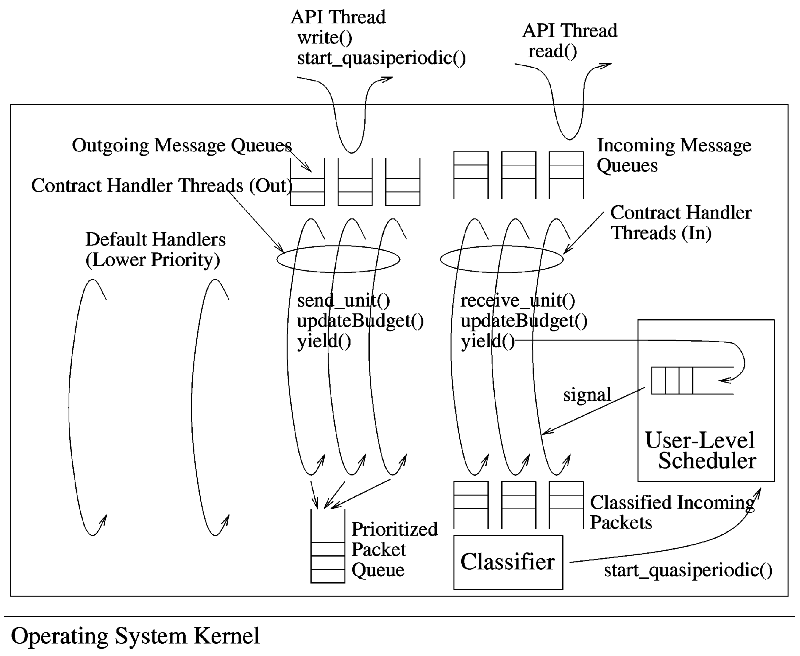


Fig. 3. User-space QoS enforcement.

and an eligibility flag F_i. The budget vector is replenished every period P_i by a periodic timer event. The eligibility flag determines whether periodic execution of the thread is enabled or disabled. It can be set or reset by the API calls $start\_quasiperiodic$ and $stop\_quasiperiodic$, respectively.

Initially, the scheduler starts out with an empty set of quasi-periodic threads. When a new client subscribes to the service and a contract C_i is established by the subscription agent (e.g., a new on-demand movie is requested), a fixed-priority kernel thread T_i is created to serve the contract. The thread registers itself with the communication subsystem scheduler, at which point a quasi-periodic thread data structure Q_i is created for it. Upon registration with the scheduler, the thread T_i is blocked on its kernel semaphore S_i. Once the eligibility flag is set (e.g., by the first frame of the transmitted video), the semaphore S_i will be signaled periodically with period P_i that is determined by the current QoS-level of the contract. The thread will be allowed to execute within each period only until one or more elements in its budget vector, E_i, expire.

An event requesting service from the contract handler thread will use the available API to start the quasi-periodic thread, which turns on the eligibility flag. If the thread has no expired resource budgets, it will be put in the ready queue of the communication subsystem scheduler immediately. Otherwise, it will be put in the ready queue when the budgets are replenished.

Since the communication subsystem scheduler does not have its own threads, but rather "borrows threads from the kernel," a kernel thread registered with it may upset its scheduling if it blocks on kernel semaphores. This is because, unless the blocking thread notifies the user-level scheduler that it is about to block, the scheduler does not know it and will not schedule another thread for execution. Any semaphore operations of quasi-periodic threads, therefore, have to use user-level semaphores. We implemented our own $semCreate$, $semWait$, and $semSignal$ operations such that blocked threads are awakened in priority order. The priority queue of the semaphore allows semaphore operations to obey the QoS-level aware prioritization policy implemented in the communication subsystem scheduler.

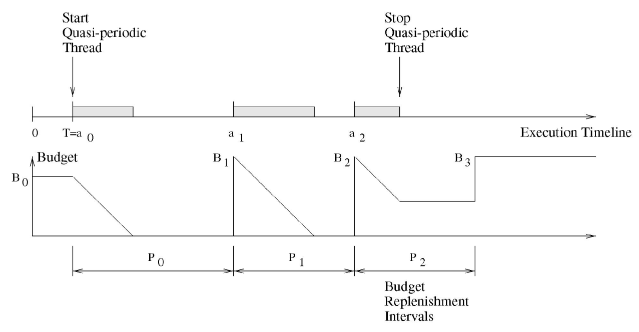
In the communication subsystem server, each $write()$ socket call wakes up an API thread, which queues up the message for transmission by the corresponding contract handler. All traffic is handled by the default contract until an explicit $socketBindContract$ call is made. The call binds a socket to a specified (nondefault) QoS contract, essentially performing classification. From then on, all communication via this socket will be deposited into that contract's message queue and processed by its corresponding contract-handler thread. The contract handler is signaled when its message queue becomes nonempty. It disables its own periodic execution when it has drained the queue. Thus, quasi-periodic contract handler threads of inactive contracts do not consume extra resources. Fig. 4b presents the communication subsystem architecture integrating QoS mapping, admission control, QoS-level selection, scheduling, contract-handler threads, traffic classification, message queues, and the outgoing packet heap.

To enforce resource allocation, the communication subsystem must ensure that the contract-handler thread for C_i gets the utilization $U_i^j[k_i]$ of each resource j, as computed by the QoS-level selection algorithms described in this paper. This requirement is viewed as a scheduling constraint. To meet this requirement, the quasi-periodic contract-handler thread for C_i is set such that its period $P_i = P_i[k_i]$ and its budget elements $E_i^j = U_i^j[k_i]P_i[k_i]$ during that period. At the end of processing a unit of service (such as transmitting a video frame or sending a URL), the server informs the communication subsystem (via a special API call), which in turn decrements each budget E_i^j by $a_j + b_j x$; the resource consumption attributed to the processed service unit (a_j and b_j are the profiling parameters obtained from QoS mapping, as described in Section 4.1).[1] In a typical contract where QoS is defined on aggregate flows, E_i^j will be sufficient to transmit hundreds of packets or serve hundreds of URLs.

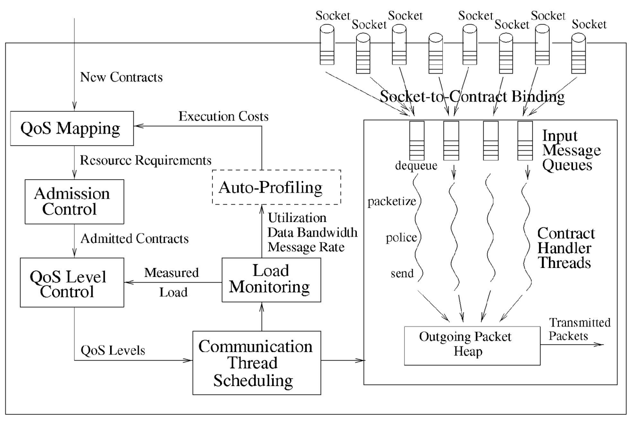
5.2 Evaluation of the OS Extensions

In order to evaluate the efficacy of the OS extensions in achieving proper resource reservation and policing for QoS guarantees, we conducted two sets of experiments. In the first, we used a best-effort version of the multithreaded communication server. In the second, we used a communication server fitted with the

1. Note that this is different from processor capacity reserves [14] where budgets represent CPU cycles only.

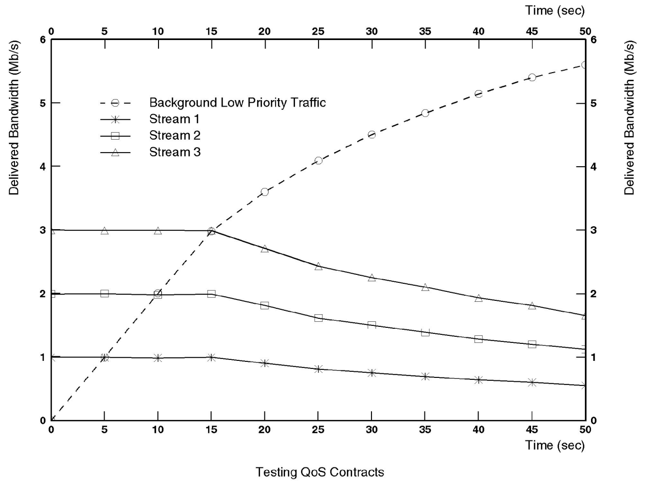


(a)

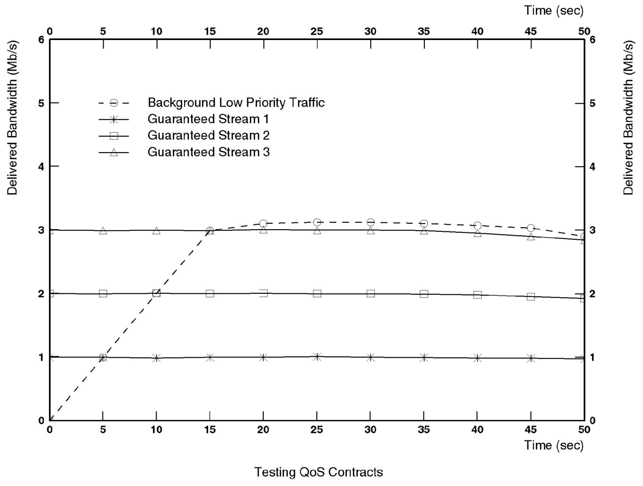


(b)

Fig. 4. Communication subsystem architecture. (a) Quasi-periodic thread execution. (b) The communication server.



(a)



(b)

Fig. 5. Enforcing QoS guarantees. (a) Best-effort server. (b) QoS-sensitive server.

aforementioned QoS support. Fig. 5 compares the resulting performance when three premium UDP flows of fixed bandwidth $1Mb$, $2Mb$, and $3Mb$, respectively, are sent concurrently with a best-effort UDP flow of gradually increasing bandwidth. Since, in this section, we are interested in evaluating enforcement (rather than utility optimization), only one QoS level was defined in each contract. This nonflexible QoS specification imposes maximum stress on the enforcement mechanism. The experiment was conducted on a Pentium PC connected to a $10Mb$ Ethernet. Once the aggregate outgoing flow saturates the Ethernet link, the best-effort server is unable to guarantee bandwidth allocation for premium flows. This effect is demonstrated in Fig. 5a by the decline in premium flow bandwidth after the communication link gets saturated. The QoS-enabled server, on the other hand, is able to provide and preserve bandwidth guarantees to premium flows, as shown in Fig. 5b. The best-effort flow occupies the remaining bandwidth.

The primary mechanisms by which QoS-sensitive performance is achieved are proper policing and protection of premium traffic from nonguaranteed traffic. Fig. 6 demonstrates the testing of these mechanisms. In this experiment, we established a QoS contract for the guaranteed traffic class A. Two resources were considered by profiling, namely, CPU consumption and communication bandwidth utilization. Two application threads were created. One sends "guaranteed" traffic through a socket bound to class A, and the other sends "nonguaranteed" traffic. Except for the socket used, the code of the two threads was identical. Each thread implemented a busy loop sending outgoing traffic. Traffic of both flows was policed to the limit shown by the dotted line in Fig. 6. The figure plots the packet rate received by each client. As shown in the figure, neither of the flows ever exceeds the policed limit, which demonstrates the correctness of the policing mechanism. Furthermore, when the network saturates at around $700pkts/s$ the lower priority flow drops as the high priority flow continues to increase, making the sum of the two flows constant and equal to the maximum packet rate that saturates the network. This demonstrates the correctness of flow prioritization. Policing and prioritization coupled with proper QoS mapping and admission control that keeps $\sum_i U_i[k_i] < 100\%$ ensure satisfaction of QoS guarantees for contracted traffic.

5.3 Portability Considerations

In this section, we discuss some considerations in implementing the QoS Contract abstraction on operating systems with no thread support, such as classical Unix. Unlike the thread-based implementation where the processing of contracted communication

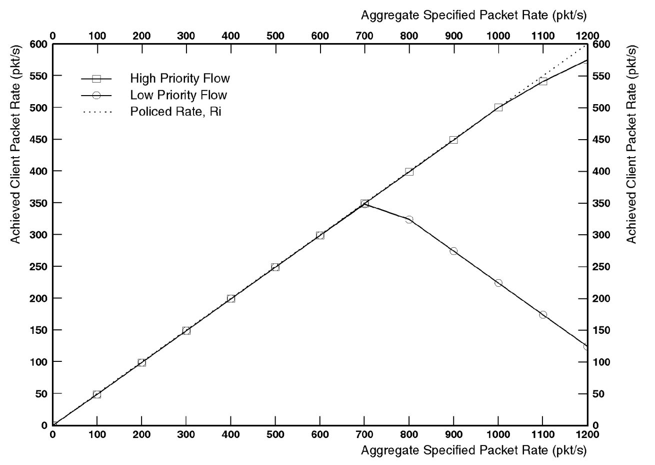


Fig. 6. Traffic prioritization.

flows is triggered by its own thread scheduler, *all* processing in the threadless implementation occurs either in the context of library calls made naturally by the application, namely, the socket library calls, or in the timer event. This approach introduces two new challenges. First, socket library calls are invoked with the granularity of request or frame processing, which is coarser than the granularity at which the communication scheduler is invoked. Second, we no longer have dedicated contract-handler threads with preassigned utilization budgets to police server traffic since our code executes directly in the context of application code.

Coarser granularity means that contracts' rate and bandwidth requirements are met when averaged on a larger time scale (e.g., 1 or 2 seconds). The lack of contract-handler threads is countered by separating the notion of a budget from the notion of a thread. In this implementation, periodically replenished per-contract budgets are maintained in order to regulate resource consumption. These budgets, however, are no longer associated with a particular thread. Hence, the multithreaded communication server is replaced with a shared memory segment that keeps track of individual contract budgets. We enable an arbitrary application process to charge "chunks" of its execution to an arbitrary contract's budget depending on the request being served. This is done via a $contractChargeBudget(C_i, x)$ call, which decrements (in shared memory) the budget for each resource j of the contract by $a_j + b_j x$, as in the previous implementation. When a budget expires, the call may be blocking or nonblocking (in which case, it returns failure). Another call, $contractCheckBudget(C_i)$ may be used to determine if any resource budget of a contract has expired. Below, we discuss how these calls may be used by different applications to achieve QoS contract guarantees.

5.3.1 Per-Flow Contracts

In applications with per-flow contracts, such as video-on-demand servers, QoS can be controlled by policing the outgoing flow (e.g., movie transmission). In such servers, the $write()$ socket library call may be instrumented to call the blocking version of $contractChargeBudget(C_i, frame\_size)$ upon each frame transmission. The call will block when some resource budget expires and will unblock it when budgets are replenished. Thus, while the communication subsystem in this case remains unaltered, the total volume pumped through any given connection is bounded by the contract.

5.3.2 Aggregate Flow Contracts

If server responses are short, contracts are more meaningfully defined on flow aggregates as, for example, is the case with web hosting applications. QoS control is best achieved by admission control applied to incoming server traffic. Admission control is achieved by instrumenting the server's $read()$ socket library call to invoke $contractCheckBudget(C_i)$ as each request is read in. The latter call returns an error if some budget of the particular contract has expired, in which case, the instrumented code will discard this request (for violation of the contracted rate). In addition, the $write()$ socket library call that sends the response to the client is instrumented to call a nonblocking $contractChargeBudget(C_i, x)$ upon response transmission to maintain an accurate budget balance. In both of the above contract types, a periodic timer replenishes the budgets and signals any blocked processes to resume. The contracted rates are therefore satisfied.

6 Related Work

Recently, QoS provisioning for Web, multimedia, and soft real-time applications has received considerable attention [6]. Since QoS provisioning is closely related to proper resource allocation and scheduling, many research efforts have focused on operating system design. For example, lazy receiver processing [9] suggests an efficient approach for structuring the communication subsystem in an operating system kernel. Processor capacity reserves [14] have been used in Mach as a new kernel abstraction to allocate processing capacity for multimedia applications [12]. Flexible CPU reservations were implemented in Rialto for efficient scheduling of time-constrained independent activities [11]. Resource containers [7] were proposed for server applications to decouple server protection domains from resource principals from the operating system's perspective. QoS-guaranteed protocol stack implementations in the user space have been proposed in [10], [13]. Our architecture differs in that it supports contracts with multiple QoS-levels that can be dynamically recomputed. In addition, our architecture does not require modifications to the operating system kernel.

Our architecture uses the QoS contract model we suggested in [4] in the context of a a QoS negotiation framework that attempts to maximize system utility. This work was extended for communication-oriented applications in [5], which advocated a new architecture for OS communication subsystems. We also presented in [3] a middleware solution for operating systems without kernel thread support. In this paper, we focus explicitly on transparency, describe for the first time our implementation of quasi-periodic threads, and establish new results regarding near-optimality of simple QoS level selection policies. Our communication architecture introduces new programming abstractions that encourage a QoS-sensitive application design methodology, yet does not preclude reusing existing server code in new QoS-sensitive contexts.

7 Conclusions

We proposed a new architecture and structuring methodology for server-side communication subsystems which supports the concept of QoS contracts. A QoS contract specifies acceptable QoS levels along with their utility and a QoS-violation penalty. In our architecture, QoS contracts are established transparently to the application server by a separate entity called the subscription manager, which makes it easier to retrofit the architecture into legacy software. We addressed the problem of optimizing aggregate user-perceived service utility on server end-systems and compared the optimal policy with simple reservation-based solutions. An implementation of utility-maximizing QoS management is presented, relying on proper resource allocation, budgeting, and policing mechanisms within the common socket API. The abstraction of dedicated contract-handler threads, called quasi-periodic threads, was discussed. In summary, communication servers which support QoS contracts are an important component of future QoS-aware services. This paper proposed new foundations for their design.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the US National Science Foundation under grant nos. EIA-980620 and CCR-0093144 and US Defense Advanced Research Projects Agency grant N00014-01-1-0576.

REFERENCES

[1] T. Abdelzaher, "An Automated Profiling Subsystem for QoS-Aware Services," *Proc. Real-Time Technology and Applications Symp.*, June 2000.

[2] T. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," *Proc. Int'l Workshop Quality of Service*, June 1999.

[3] T. Abdelzaher and K.G. Shin, "QoS Provisioning with *q* Contracts in Web and Multimedia Servers," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1999.

[4] T.F. Abdelzaher, E.M. Atkins, and K.G. Shin, "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control," *Proc. IEEE Real-Time Technology and Applications Symp.*, June 1997.

[5] T.F. Abdelzaher and K.G. Shin, "End-Host Architecture for QoS-Adaptive Communication," *Proc. IEEE Real-Time Technology and Applications Symp.*, June 1998.

[6] C. Aurrecoechea, A. Cambell, and L. Hauw, "A Survey of QoS Architectures," *Proc. Fourth IFIP Int'l Conf. Quality of Service*, Mar. 1996.

[7] G. Banga, P. Druschel, and J.C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, chapter 25, pp. 527-531. The MIT Press, 1990.

[9] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," *Proc. Symp. Operating Systems Design and Implementation (OSDI)*, 1996.

[10] R. Gopalakrishnan and G. Parulkar, "Efficient User Space Protocol Implementations with QoS Guarantees Using Real-Time Upcalls," *IEEE/ACM Trans. Networking*, 1998.

[11] M. Jones, D. Rosu, and M.-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. 16th ACM Symp. Operating Systems Principles*, Oct. 1997.

[12] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach," *Proc. Multimedia*, Mar. 1996.

[13] 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.

[14] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems*, May 1994.

Thinning Schemes for Call Admission Control in Wireless Networks

Yuguang Fang, *Senior Member*, *IEEE*

Abstract—In this paper, we present new call admission control schemes, *the thinning schemes*, which smoothly reduce the traffic admission rates. Performance analysis is carried out and new analytical results are obtained. It demonstrates that the thinning schemes can be used to derive many known call admission control schemes.

Index Terms—Call admission control, Resource allocation, Wireless networks, Multimedia, Blocking probability.

✦

1 INTRODUCTION

THE future telecommunications networks (such as the third generation wireless networks) target providing integrated services, such as the voice, data, and multimedia, via inexpensive low-powered mobile computing devices over the wireless infrastructures ([1], [2]). The demand for multimedia services over the air has been steadily increasing over the last few years, leading to the design consideration of wireless Internet. Depending on the QoS (Quality of Service) requirements for various service requests from mobile users, different priorities may be assigned to various call connections. For example, real-time services such as voice or streaming videos may be assigned higher priority over non-real-time services such as data; handoff call connections should be given higher priority over new call connections in order to reduce the call dropping probability; mission critical data should be handled with higher priorities than some real-time data such as voice; users who pay more for their services should be treated with higher priorities over those who pay less. In order to support such mixed service requests in these wireless networks with multiple traffic types, efficient resource provisioning is a major issue ([2], [3]). Call admission control (CAC) is such a provisioning strategy to limit the number of call connections into the networks in order to reduce the network congestion and call dropping probabilities.

Prioritized traffic systems consisting of new calls and handoff calls in wireless networks have been intensively investigated in the literature (see [4], [5] and references therein). An admitted call for a mobile user may have to be handed off to another cell into which the mobile user moves, hence the call may not be able to gain a channel in the new cell due to the limited resource in wireless networks, which will lead to the call dropping. Thus, new calls and handoff calls have to be treated differently in terms of resource allocation. Since users tend to be much more sensitive to call dropping than to call blocking, handoff calls are normally assigned higher priority over new calls. The guard channel scheme ([3]) has been proposed to handle such systems: A proportion of the channels assigned for a base station has been reserved for handoff calls. This guard channel scheme can be generalized to handle the multimedia networks with multiple classes of priority services. Li et al. ([4]) have studied the guard channel scheme for wireless networks with multiple traffic types, the *multiple thresholding scheme*, in which different thresholds are used for each traffic type,

• *The author is with the Department of Electrical and Computer Engineering, University of Florida, 435 Engineering Building, PO Box 116130, Gainesville, FL 32611. E-mail: fang@ece.ufl.edu.*

Manuscript received 20 Sept. 1999; revised 14 June 2001; accepted 14 May 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110621.