End-host Architecture for QoS-Adaptive Communication *

Tarek Abdelzaher and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*zaher, kgshin*}@*eecs.umich.edu*

Abstract

Proliferation of communication-intensive real-time applications with "elastic" timeliness constraints, such as streaming stored video, requires a new design for end-host communication subsystems. The design should (i) provide per-flow or per-service-class guarantees, (ii) maximize the aggregate utility of the communication service across all clients, (iii) gracefully adapt to transient overload, and (iv) avoid, if possible, starving lower-priority service classes during the period of sustained overload.

We propose a QoS-optimization algorithm and communication subsystem architecture that satisfy the above requirements. It provides each client its contracted QoS, while adapting gracefully to transient overload and resource shortage. A new concept of flexible QoS contract is introduced, specifying multiple acceptable levels of service (or QoS levels for short) and their corresponding rewards for each client. Allowing clients to specify multiple QoS levels permits the server to perform QoS-optimization and degrade client's QoS under transient overload predictably, as specified in the QoS contract. Clients receive a money-back guarantee if the contracted QoS is violated by the server. The proposed resource-management mechanism maximizes server's total reward under resource constraints. We implemented and evaluated the architecture on a Pentium-based PC platform running under The Open Group (TOG) MK7.2 kernel, demonstrating the capability of our communication subsystem in meeting its design goals.

*The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-94-1-0229 and Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Rome Laboratory under Grant F30602-95-1-0044. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the funding agencies.

1. Introduction

This paper presents a new communication subsystem architecture for QoS-adaptive real-time applications. The main contribution of this architecture is its dynamic QoS-optimization mechanism and support for flexible QoS contracts. The QoS contract between the communication client and server specifies not only the QoS requirements of the nominal mode of operation (or nominal *QoS level*), but also degraded acceptable QoS levels. Thus, under overload, the communication subsystem can degrade connections *predictably* as specified in their respective QoS contracts. In order to express the relative desirability of the acceptable QoS levels specified in a QoS contract, the contract associates a *reward* with each level. The reward may be commensurate with the client-perceived utility of receiving service at that level. Ideally, each client receives service at the maximum QoS level specified in its QoS contract. To achieve graceful degradation, under overload a *QoS-optimization algorithm* determines how to degrade individual clients. Thus, instead of inflicting arbitrary degradation, the aggregate system bandwidth is subdivided among clients so as to optimize a selected performance measure. If it is impossible to service the client even at the lowest acceptable QoS level, the client must be "reimbursed" for his broken contract. The QoS contract specifies a *QoS-violation penalty* to be paid by the service provider in such a case. Resource management ensures that clients will either receive their contracted service (and get "billed" for the given QoS level), or receive a "rebate" corresponding to the specified QoS-violation penalty. The goal of the server is to maximize its reward. The QoS-optimization algorithm chooses clients' QoS levels that optimize the aggregate reward subject to resource constraints, given the information on rewards and QoS-violation penalties.

Our work is complementary to the research on real-time communication protocols [2, 10, 18, 21], network support for QoS [6, 19], and real-time operating system support [9, 12, 15, 17]. We address end-host design that supports a QoS-adaptive service model [1, 4, 7, 8]. Unlike integrated end-to-end architectures geared towards multimedia QoS [3, 5, 16, 20], we focus on providing appropriate QoS guarantees for both soft and hard real-time applications by the composition of individually-designed independent components. In this approach the desired end-to-end behavior is achieved by composing several independent resource-management modules (for computing and network resources), each appropriate for the particular resource it manages and the particular environment in which it executes. For this composability, no built-in assumptions should be made in a module regarding others with which it might be composed. In this paper, we focus on the *end-host QoS-management module* only. To achieve its independence from other modules in an end-to-end scheme our design makes no assumptions about the communication protocol stack, type of network, resource-reservation support in the network, OS support for QoS on the server, and does not require exact system-load characterization and profiling.

Ideally, QoS adaptation should be coordinated at both ends and all internal/intermediate points of the connection. Real-time communication protocols such as RTP [18], can be used with our scheme to help clients adapt application-level performance to network delays. RSVP [21], or real-time channels [2, 10] can be used to reserve host and network bandwidths, when applicable. Further development of such techniques for flexible end-to-end guarantees is beyond the scope of this paper. We design an end-host communication subsystem to meet the following goals.

- Provide per-flow or per-service-class guarantees on the end-host. Once a QoS contract is established with a client, it will be enforced.
- Maximize the aggregate reward (for delivered QoS) across all clients. In applications such as video-on-demand servers where clients pay for their received QoS, this maximization translates into maximization of the total server's revenue.
- Adapt responsively to transient load fluctuations and resource shortage. This is in sharp contrast to indiscriminate degradation that applies to premium-service clients as well as to basic-service clients alike.
- Avoid starving lower-priority service classes under prolonged overload conditions, if possible. Depending on how reward values and QoS-violation penalties (rebates) are set, it may be more profitable to "mildly" degrade the performance of some set of clients rather than to discontinue the service completely. Under overload, our algorithm degrades clients in a way that optimizes the reward; this is in sharp contrast with simple fixed priority schemes that will starve lower-priority clients.

Section 2 defines the proposed new service model and present the corresponding communication subsystem architecture. The main components of the architecture are elaborated on in Sections 3 and 4, respectively. Section 5 details our implementation of the adaptive communication subsystem. Evaluation and testing results are presented in Section 6. The paper concludes with Section 7.

2. Real-time Communication Model

A real-time communication subsystem must enforce the semantics of its QoS contract with clients. We now elaborate on these semantics and present the general architecture of the communication subsystem.

2.1. QoS Levels

In order to provide a predictable communication service, the amount of communication traffic generated at the source must be bounded. The leaky bucket model [11] has been commonly used to control input traffic. Thus, at the communication subsystem level, each QoS level may be represented by a leaky-bucket-like traffic specification of different size and rate. From a resource-management viewpoint, the abstraction is somewhat similar to a resource-capacity reserve. A QoS level can be viewed as a communication-capacity reserve C_i, specified by a number M_i of bytes,[1] a period P_i (the inverse of rate) and a buffer size B_i. The reserve can be of *sender* or *receiver* type. At the sender side, the reserve holds enough computing and buffer resources for M_i bytes to be transmitted every P_i units of time. The buffer-size parameter specifies the maximum number of bytes to be buffered due to the sender's burstiness. Without loss of generality, we can assume that B_i is specified in multiples of M_i. The maximum number $N_{max}(t)$ of bytes that can be generated within an interval of length t is $(B_i + \lfloor t/P_i \rfloor)M_i$. Since $\lfloor t/P_i \rfloor M_i$ of these bytes will have been transmitted by time t, the expression for $N_{max}(t)$ implies that at most $B_i M_i$ can be awaiting transmission at the sender at any given time. A byte generated at time t at the source is said to be *conformant* if the total number of bytes generated by time t is less than $N_{max}(t)$. Guarantees

[1] M_i can be specified in bytes if messages are much larger than the maximum packet size. Otherwise, it is specified in packets. In any case we should be able to estimate (approximately) an average number of packets given M_i.

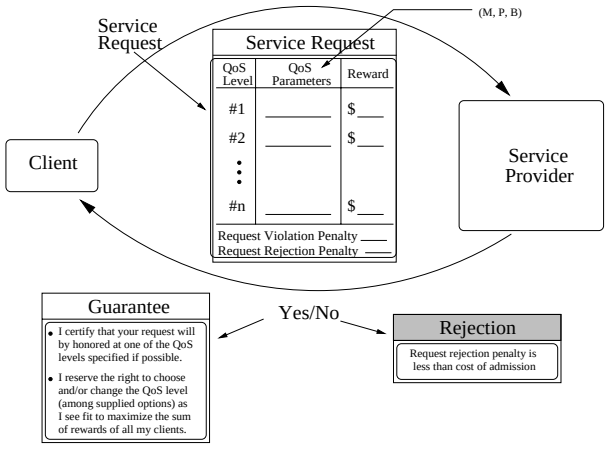

Figure 1. Connection-establishment request

are given to conformant bytes only. Since M_i bytes are guaranteed to be transmitted every P_i units of time, a conformant byte is guaranteed to be transmitted within $B_i P_i$ time units after its generation at the source. Thus, the transmission deadline of a byte generated at time t is $D_i = t + B_i P_i$.

At the receiver the QoS level is a communication-capacity reserve that holds enough computing resources to receive M_i bytes every P_i time units. The buffer size parameter B_i specifies the maximum size of the delivery buffer, measured in multiples of M_i. This parameter is useful when the receiver task is not synchronized with the communication service, in which case the service must buffer incoming packets until the application executes a receive. A byte is said to have been *delivered* when it has been processed by the protocol stack at the receiver and deposited in a delivery buffer.

2.2. The QoS Contract

A QoS contract is signed between the server and client upon connection establishment. The QoS contract specifies alternative *QoS levels* acceptable to the client, their corresponding *rewards*, and a *QoS-violation penalty*. Each QoS level is expressed in terms of its M, P, and B traffic parameters described in Section 2.1. Technically, the server may also specify a QoS-*rejection* penalty incurred if a new request is rejected prior to contract establishment (as opposed to the QoS-violation penalty which represents the cost of violating an existing contract). For many applications one may argue that there should be no rejection penalty since no contract has yet been established. Figure 1 shows an example connection-establishment request. Should the QoS contract be "signed" by the server, the client is guaranteed to receive the service at *one* of its requested QoS levels, or be "paid" the specified amount of QoS-violation penalty. The selection of a particular QoS level is delegated to the server (in this case, the communication subsystem). Furthermore, the communication subsystem may change the delivered QoS to another level in the client's QoS contract at its discretion, when appropriate, without consulting the client. This gives the subsystem enough leeway for QoS adaptation and the optimization of aggregate reward.

Note that specifying only one acceptable QoS level with a default (e.g., infinite) violation penalty reduces the QoS contract to a traditional on-line admission control API. An infinite violation penalty would mean that a QoS violation results in a system failure, as is the case in hard real-time systems.[2] If such is the case, the underlying system should also support resource reservation, and the worst-case resource requirements of arriving requests should be known upon their arrival. Since we are primarily interested in applications which have flexible QoS requirements, we do not require resource-reservation support in the general case.

2.3. Reward Estimation

The problem of specifying rewards for different QoS levels (as well as the QoS levels themselves) warrants additional discussions. In those real-time applications interacting with a human user (rather than a physical environment), QoS levels and rewards are somewhat subjective. For example, the parameters of QoS levels may be set by application designers to correspond to "poor", "good" and "excellent" performance, respectively. The reward values may be set by the end users at the time of their service requests using an intuitive GUI, such as a slide rule which quantifies user satisfaction with each of the above-predefined QoS levels on a 0–100% scale. In a commercial, however, QoS levels and rewards may be selected by the service provider. For example, a communication server serving multiple human users can use reward information based on a combination of customized user preferences and user membership fees.

QoS-level and reward specification for applications that deal with an external physical environment, as opposed to a human user, should be done based on an objective performance measure. For example, in [1] we illustrated a case where QoS levels and rewards were specified by an AI agent representing an application domain expert for an automated flight application to minimize the mission failure probability. In general, the specification of QoS levels and rewards is an application-specific policy that should be discussed in the context of the particular application. Our main goal in this paper is to provide general mechanisms for such specification that are rich enough to express elas-

[2] Indeed, incurring an infinite penalty makes it meaningless to maximize the accrued reward.

tic service requirements, whenever such information is available.

2.4. System Architecture

This subsection highlights the basic architectural elements used to satisfy our communication subsystem design requirements. The first and most basic objective of our architecture is to provide per-client or per-service-class QoS on the server. For this, each client (or service class) is handled within the communication server by a *separately schedulable* entity. We will henceforth call it an *adaptive negotiation agent* (ANA). An ANA is created at the time of connection setup within a communication server on behalf of an application client. It expresses the client's QoS contract terms to the communication server and is scheduled in accordance with the client's assigned QoS level. Associating different ANAs with different clients allows clients to be serviced concurrently at different QoS levels.

The second objective of our architecture is to maximize the server's aggregate reward. This is accomplished by a proper choice of QoS levels that "optimally" utilizes available resources. The reward maximization problem can be translated into that of proper ANA scheduling and schedulability analysis. At any given time, the communication subsystem contains a set of ANAs, each scheduled at a particular QoS level specified in its QoS contract. To maximize the aggregate achieved reward subject to resource constraints, our design utilizes a combination of (i) a load control module implementing a QoS-optimization algorithm that maximizes aggregate reward using an estimate of current load, and (ii) a monitoring module that measures and updates estimated load parameters. The load control module is activated when a new ANA is to be created or destroyed. Upon activation, it (i) performs an admission test on the new ANA, if any, and (ii) recomputes the active QoS level for all current ANAs to adapt to the current load. The amount of resources available to the communication subsystem as a whole may be fixed (dedicated resources), or may vary dynamically (e.g., in the absence of appropriate resource-reservation mechanisms) in which case the load control module may also be executed periodically to adapt to dynamic load changes. Due to the relative complexity of QoS optimization, the invocation period of the load control module must be relatively slow. The module is described in more detail in Section 3.

The third objective of our architecture is to adapt responsively to transient load or resource-capacity changes. Such changes can occur between successive invocations of the load control module and may require immediate response. Since re-executing the QoS-optimization algorithm upon every load change may consume too much bandwidth, we provide a fast feedback loop, implemented by the monitoring module, that detects transient overload or underutilization conditions and makes small incremental adjustments to the QoS levels selected by load control using a simple heuristic as described in Section 4. Together the periodic optimization and the fast QoS level adjustment heuristic ensure that during overload clients are gracefully degraded, but not starved, whenever such degradation leads to a higher aggregate achieved reward. This feature satisfies the fourth objective of the architecture. Figure 2 depicts the major components of the communication subsystem.

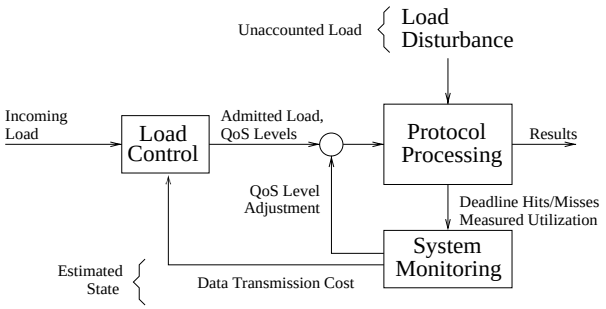

Figure 2. The communication subsystem

3. Load Control

In this section, we describe the communication subsystem component responsible for admission control and near-optimal selection of QoS levels for ANAs such that the aggregate achieved reward is maximized. We first consider a special case where the computing power and communication bandwidth available to the communication subsystem are fixed. This may be the case with hard real-time systems where resources are dedicated *a priori*. We then extend the result to the case where these parameters vary dynamically.

3.1. QoS-Optimization

Assume that the communication subsystem is allotted a fraction U_{server} of CPU utilization, and that the host has an amount BW of network bandwidth dedicated to it. Upon invocation, the load control module executes a QoS-optimization algorithm that selects QoS levels to maximize the aggregate reward subject to the aforementioned resource constraints. Let there be n ANAs considered by the load control module (including any newly-arrived ANA-creation request). Let the QoS contract Q_i of ANA$_i$ contain m_i QoS levels, denoted by $L_i[1], ..., L_i[m_i]$, let their rewards be $R_i[1], ..., R_i[m_i]$, respectively, and let the QoS-violation penalty be V_i. As mentioned in Section 2.1 each QoS level $L_i[k]$ is given by a maximum data size $M_i[k]$, a period $P_i[k]$,

and a buffer size requirement $B_i[k]$. At this point we introduce an additional (artificial) QoS level for each ANA, called the *null level* (at which the client receives no service), with the cost of starving or rejecting the client. The level has no resource requirements (i.e., $M_i[k] = 0, P_i[k] = \infty, B_i[k] = 0$), and a negative reward numerically equal to its QoS violation penalty (i.e., $R_i[k] = -V_i$) if the client has a contract, or its QoS-rejection penalty if it is a new client being considered for admission. Let $f_c(x)$ be the cost function which gives the total time it takes to process and transmit x bytes. Let's define the periodic execution cost, $E_i[k]$, such that:

$$E_i[k] = f_c(M_i[k]). \quad (1)$$

The processor utilization requested by a QoS level $L_i[k]$ is:

$$U_i[k] = E_i[k]/P_i[k]. \quad (2)$$

This utilization must be less than U_{server} for the QoS to be feasible. Furthermore, the requested network bandwidth, $BW_i[k]$, is $M_i[k]/P_i[k]$ which must not exceed the available bandwidth BW. The QoS-optimization problem is NP-complete because it is a derivative of the multiple-choice knapsack problem. To reduce the NP-complete optimization problem to a polynomial-time problem, we assume that $U_i[k]$ can only take discrete values which are multiples of some small constant δ. In practice, this means that the actual utilization allotted by the communication system to an ANA will always be approximated to the next multiple of δ. The value of δ can be chosen appropriately small. Now, let's define $S(i,U)$ to be the subproblem of optimally assigning QoS levels to ANA_1, ..., ANA_i without exceeding some arbitrary utilization U of the computing resource. For notational simplicity, let $S(i,U)$ also denote the corresponding aggregate reward. Let $BW(i,U)$ represent the corresponding network bandwidth consumed. Furthermore, let $S(i,*)$ be the set of subproblems $\{S(i,\delta), S(i,2\delta), ..., S(i,U_{server})\}$. Given n ANAs in the subsystem we need to solve the problem $S(n,U_{server})$. Using the dynamic programming the following recursive relation is derived:

$$S(i,U) = \max_{1\le k\le m_i}\{R_i[k] + S(i-1, U-U_i[k]) \mid BW_i[k] + BW(i-1,U-U_i[k]) \le BW\} \quad (3)$$

Note that if k^* is the value of k in the above equation that yields the computed maximum, then:

$$BW(i,U) = BW_i[k^*] + BW(i-1, U-U_i[k^*]) \quad (4)$$

For the special case of $i = 1$,

$$S(1,U) = \max_{1\le k\le m_1}\{R_1[k] \mid U_1[k] \le U, BW_1[k] \le BW\} \quad (5)$$

and if k^* is the value of k that yields the maximum in the above equation:

$$BW(1,U) = B_1[k^*] \quad (6)$$

The above recursive relation computes the solution to a subproblem in $S(i,*)$ given solutions to the subproblems in $S(i-1,*)$, taking one $O(m_i)$ step. Note, however, that since the utilization is discretized, there is only a finite constant, K, of possible utilization values in the range $[0, U_{server}]$, i.e., $|S(i,*)| = K$, for $i = 1, \ldots, n$. Thus, there are a total of only nK subproblems $S(i,U)$ to be solved. Solving all of these subproblems (in increasing order of index i) will therefore take $O(K\sum_{1\le i\le n} m_i)$ which is equivalent to $O(KnL_{av})$, since $L_{av} = (\sum_{1\le i\le n} m_i)/n$. Finally, note that K is a constant that does not grow with problem size, making the complexity of the algorithm $O(nL_{av})$. The complete algorithm is given below.

```
Algorithm A
1 for i = 1 to n
2   for U = 0 to U_server in steps of δ
3     compute S(i,U) from Equation 3
4     compute BW(i,U) from Equation 4
5 return S(n, U_server)
```

The above algorithm exhaustively searches the space of all solutions in the discretized CPU utilization space and returns the optimal one. The returned solution assigns a QoS level for each ANA such that the aggregate server reward is maximized. From a scheduling perspective, the QoS level determines the period at which each ANA is to be invoked. From the ANA's perspective, the QoS level determines the amount of data to be processed per period, and the size of the data buffer.

3.2. Practical Approximation of QoS-Level Selection

The algorithm presented above is a multiple-choice knapsack problem with the CPU being the knapsack, and network bandwidth requirements being an additional constraint. From a mathematical perspective, the CPU and network bandwidth can be viewed as two abstract resources. If the knapsack problem is solved for the *bottleneck* resource (of these two), the constraint on the other resource will always be satisfied. In our architecture, CPU processing at the host and packet transmission by the network device occur concurrently for successive packets as a two-stage pipeline. The cost, $f_c(x)$, of transmitting x bytes is therefore the cost of the bottleneck stage. We lump the components of $f_c(x)$ into a per-packet processing cost and use a linear approximation $f_c(x) = \gamma\lceil x/MTU\rceil$, where γ is an estimated per-packet processing cost of the bottleneck stage, and MTU is packet size. Using Equations 2,

and 1, the bottleneck resource utilization requested by a QoS level $L_i[k]$ is:

$$U_i^{bottle}[k] = E_i[k]/P_i[k] = \gamma\lceil M_i[k]/MTU\rceil/P_i[k]. \quad (7)$$

Equations 3 and 5 can now be rewritten for the bottleneck resource by substituting in them the above utilization and omitting the always-satisfied constraint on the non-bottleneck resource.

$$S(i,U) = \max_{1\le k\le m_i}\{R_i[k] + S(i-1, U - U_i^{bottle}[k])\} \quad (8)$$

$$S(1,U) = \max_{1\le k\le m_1}\{R_1[k]\} \quad (9)$$

Finally, let's examine γ, the per-packet processing cost of the bottleneck stage. If V is the observed system throughput in packets per second, and U^b_{server} is the current utilization of the bottleneck resource, then $\gamma = U^b_{server}/V$. Since U^b_{server} changes dynamically and may be difficult to measure, we scale utilization units by $1/U^b_{server}$. Thus, γ expressed in the new units, called γ_{effect}, is $1/V$. Consequently, **Algorithm A** can be rewritten in the new units as follows:

```
Algorithm A'
1 for i = 1 to n
2   for U = 0 to 1 in steps of δ
3     compute S(i,U) from Equation 8
4 return S(n, 100)
```

where the (scaled) utilization values in Equation 8 are computed from:

$$U_i[k] = E_i[k]/P_i[k] = \gamma_{effect}\lceil M_i[k]/MTU\rceil/P_i[k]. \quad (10)$$

The approximate version of the algorithm means that in soft real-time systems where (i) the allotted resource capacity U_{server} and BW are neither fixed nor known, (ii) the bottleneck resource is not known or can change dynamically depending on end-host and network load conditions, and (iii) the system is not profiled (i.e., $f_c(x)$ is not exactly specified), we can still compute near-optimal QoS levels by measuring system throughput, V, computing its inverse γ_{effect}, then applying **Algorithm A'**. In a system where resource availability changes dynamically, the computed value of γ_{effect} may change at run-time. **Algorithm A'** will be executed periodically to recompute QoS levels accordingly.

3.3. Client Aggregation

It is possible to aggregate clients into a number of generic classes. Each class may have standard predefined QoS levels and rewards. The value of $M_i[k]$ for a class C_i will be interpreted as a "budget," and may be chosen to be sufficient for serving some pre-selected number M of clients of that particular class. A client requesting connection to the server will specify the class of service it desires. The communication requirements of the client will then be debited to the ANA of the class. When enough clients request a specified class to exceed its budget, a new ANA may be created to serve the "overflow" clients in the class.

4. Monitoring and Feedback Control

In the previous section we described how QoS levels are near-optimally selected for ANAs. Since the algorithm incurs a relatively high cost, it is more suitable for adaptation to long-term resource variations. Thus, in our implementation it is executed once every 5 seconds. γ_{effect} is computed by averaging the monitored throughput V over a sufficiently large window, then obtaining its inverse. In the current implementation V is averaged over 50 readings, each collected over a $100ms$ time interval. A more responsive mechanism is needed to adapt to short-term load changes. In order to respond to fast load fluctuations, we check for transient overload or underutilization conditions every P time units, and make a small incremental adjustment to the QoS levels of some ANAs around the operating point computed by **Algorithm A'**. The purpose of QoS-level adjustment is to keep the system from getting underutilized or overloaded due to transient load or inaccurate γ_{effect}. Adjusted QoS levels are always selected among those specified in the QoS contract for the ANAs in question. Overload conditions are detected by checking whether or not scheduled ANAs fail to execute by their deadlines (end of their invocation period). When d deadlines are violated, "overload" is flagged. The parameter d can be configured according to deadline criticality. For example, if meeting every deadline is very important, then $d = 1$ is the most favorable choice. However, if a certain number n of deadline misses can be tolerated before an adverse effect is perceived then a good choice would be $d = n$. In addition, every period P, the monitoring module looks up the total time μ, out of P, during which at least one ANA was ready or running. The ratio μ/P represents system utilization by the scheduled ANAs. Underutilization conditions are flagged when μ/P drops below a pre-configured threshold.[3] Once overload or underutilization is flagged, a QoS-adjustment heuristic is executed to adapt QoS levels accordingly. It executes a fast greedy algorithm attempting to maximize the aggregate achieved reward as shown below:

[3] In our implementation a threshold of 85% utilization is used.

```
QoS Level Adjustment Heuristic
1 if overload then
2   Choose the ANA, C_i, whose degradation
    by one QoS level results
    in the minimum decrease in reward
    (i.e., R_i[current] − R_i[current − 1] is minimum)
3   Degrade C_i to next QoS level.
4 if underutilization then
5   Choose the ANA, C_i, whose promotion
    by one QoS level results
    in the maximum increase in reward
    (i.e., R_i[current + 1] − R_i[current] is maximum)
6   Promote C_i to next QoS level.
```

The algorithm will alter system load hopefully resolving the overload or underutilization conditions. In the current implementation, these conditions are checked at a 10-Hz frequency. Note that since the heuristic starts from a near-optimal solution for the QoS-level selection problem, it has a good chance of finding a global optimum.

5. Implementing QoS Contracts

A thread-per-connection model has been suggested for communication subsystem design in [14]. We extend this model to ANAs, offering a thread-per-ANA architecture. We implement a user-level thread package, called $q$threads, whose scheduler transparently performs the QoS optimization and dynamic QoS-level adjustment algorithms described in Sections 3 and 4, and assigns thread scheduling priorities appropriately to satisfy the selected QoS levels. This scheduling package is novel in that its threads are explicitly aware of their own QoS levels.

5.1. $q$threads: QoS-Adaptive Threads

The $q$thread package has been implemented with the intention of supporting generic adaptive server architectures. The package is novel in that its scheduler explicitly recognizes per-thread QoS levels, rewards, and QoS-violation penalty. This information is passed in the *qthread_create()* primitive. A QoS level $L_i[k]$ of thread T_i is expressed in terms of a period $P_i[k]$ and an execution budget $E_i[k]$ specifying that the thread must execute for $E_i[k]$ every period $P_i[k]$. This implies a requested CPU utilization $U_i[k] = E_i[k]/P_i[k]$. For architectures with a self-profiling capability, $E_i[k]$ is specified as two values. An absolute computation time $e_i[k]$ that specifies computational requirements in application-specific absolute load units, and a pointer p to a scaling factor β, presumably maintained by the self-profiling subsystem that expresses the actual execution time per unit of absolute load on the underlying platform. Thus, $E_i[k] = \beta e_i[k]$. In our use of the package $e_i[k]$ is set to $\lceil M_i[k]/MTU \rceil$, while p points to the computed γ_{effect}. When a thread is created using the *qthread_create()* primitive, QoS optimization is invoked to compute a QoS level for each thread, as described in Section 3, so that the aggregate achieved reward is maximized. The *qthread_create()* call fails if the algorithm rejects the thread. Otherwise, the thread is created and invoked periodically with a period determined by its selected QoS level. Threads are scheduled using EDF with deadline equal to thread period, which is an optimal scheduling strategy.

5.2. Implementing ANAs

Each ANA is realized as a separate $q$thread. ANA objects are implemented within a communication library called *CLIPS* (Communication Library for Implementing Priority Semantics). The main tasks performed by *CLIPS* are as follows:

QoS-level-sensitive message dequeuing: Each ANA$_i$ at the sender has an input message buffer sized to take $B_i[k]M_i[k]$ bytes, where k is the ANA's QoS level. The sending source deposits messages in that buffer while the ANA dequeues them asynchronously, every period $P_i[k]$ as specified by its QoS level. If the buffer fills up, the source is blocked by the ANA on the next message enqueue operation, until buffer space becomes available, or alternatively returns an error code. If several sources deposit messages into the same ANA (as might be the case with client aggregation, described in Section 3.3) they are assigned separate message queues in the common input message buffer, and are dequeued in a round-robin fashion to achieve fair bandwidth sharing, and guarantee each client a minimum share of the ANA bandwidth.

Packetization/Depacketization: This function is typically performed by a protocol layer in the protocol stack executed by the ANA. The ANA can be configured to execute an arbitrary protocol stack.

Per-ANA policing: In order to prevent clients from violating their traffic specification, client traffic is policed on a per-ANA basis to ensure that each connection is conformant to the rate of service dictated by its current QoS level. Traffic is policed by limiting each ANA to send (receive) no more than $\lceil M_i[k]/MTU \rceil$ packets during each period $P_i[k]$. This is achieved by counting the number of processed packets and blocking once the ANA has used its allotted packet budget for the given period, until the start of the next period. Note that such voluntary blocking (yielding) indirectly enforces a CPU run-time limit on each invocation of the ANA.

Outgoing packet queuing: The ANA, on the sender side, deposits processed packets in an outgoing queue. When the communication link becomes available to the sending host, link bandwidth must be allocated to outgoing packets in proper priority order to provide the QoS lev-

els selected by $q$threads. Outgoing packets are therefore queued at the network device interface in a priority heap sorted by ANA deadline, as assigned to it by the $q$thread scheduler.

Receive side demultiplexing: Incoming packets are demultiplexed by the receiver which then invokes the $q$thread (ANA) responsible for serving the particular connection. The communication subsystem scheduler then schedules each ANA to run in accordance with the deadline assigned to it from its QoS level.

5.3. Implementation Platform

We implemented the proposed communication architecture on a Pentium-based PC network, each running the MK 7.2 microkernel developed by The Open Group (TOG).[4] MK 7.2 is a derivative of CMU's Mach. The communication subsystem architecture was implemented using the facilities of TOG's CORDS environment which is based on x-kernel support originally developed at the University of Arizona. At present, the communication subsystem is realized as a CORDS server implementing the communication protocol stack on top of the microkernel and communicating directly with the network device driver via kernel ports.

6. Evaluation

We conducted several experiments to stress-test the performance of the communication subsystem server and verify its ability to meet the contracted QoS requirements. The server was run on a single Pentium-based PC with an MK 7.2 kernel. To emulate natural operating conditions, the machine was placed on one segment of a shared Departmental Ethernet serving 204 machines. The Ethernet is connected to a campus-wide network via an FDDI ring that is in turn connected to an Internet backbone. First, we identified the cost of QoS adaptation in the proposed architecture. Our profiling results indicate that the monitoring module overhead is about $7\mu s$, the QoS-optimization overhead is approximately $32\mu s$ per QoS-ANA, and the heuristic QoS-adjustment overhead is $8\mu s$. Currently, monitoring and heuristic QoS adjustment are performed once every $100ms$, and QoS-optimization is invoked every 5 seconds. For 10 ANAs created in the system (due to client aggregation the number of clients can be much more), the above figures indicate that the aggregate overhead consumed by QoS-adaptation mechanisms is less than 0.1 %.

To test server performance we created a set of threads, each of which generates messages persistently and sends them using our communication subsystem API. Each thread represented the endpoint of a connection. We shall henceforth call it a *generator thread*. An ANA was created for each connection to enforce the QoS contract, and police the generator. Connection throughput was measured at the bottom of the protocol stack of the sender just above the network device driver. Since we are not addressing end-to-end QoS in this paper, we ignore the receiving end in this evaluation. Figure 3 shows the measured throughput versus time for each of three connections. In this experiment the contacted rates for the connections were $1.5Mb/s$, $1Mb/s$ and $0.5Mb/s$, respectively.[5] The figure shows that although the generator threads dump messages persistently to the communication subsystem without regard to a maximum rate, their measured throughput does not violate the contracted QoS due to appropriate policing.

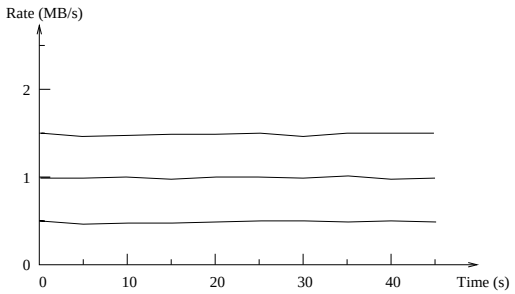

Figure 3. Policing effect

The rate-policing mechanism alone, however, is not sufficient for enforcing the QoS contract. In Figure 4 we show what happens when we increase the number of connections above the schedulable limit. In this experiment, we incrementally add 1Mb generator threads, creating a new ANA for each. The QoS contract for each connection has $M_i = 100kb$, and $P_i = 100ms$. As we can see, the system becomes overloaded and the average connection throughput decreases below its contracted value as the aggregate bandwidth consumed by the system saturates. In our experiments the maximum aggregate consumed bandwidth was found to be approximately $3.7Mb/s$. The inability of individual connections to receive their contracted rate calls for an admission-control mechanism to ensure that the set of all connections is schedulable.

We incorporated the admission-control algorithm presented in Section 3.2. The algorithm uses an approximate estimate of the transmission-cost-per-packet γ_{effect} (which it gets from run-time monitoring) to compute the maximum allowable load. The on-line estimated value of γ_{effect} was about $3.25ms/pkt$, which yields a maximum throughput of about 307 (Ethernet) packets per second. This throughput permits admitting only three $1Mb$ connections. From Figure 4 we

[4] TOG is formerly known as the Open Software Foundation.

[5] where b, in this section, refers to *bits*.

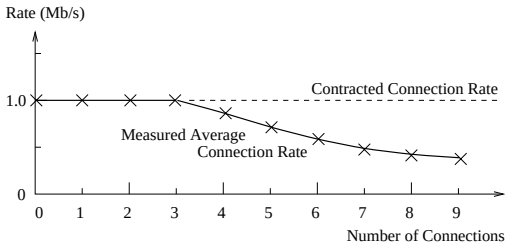

Figure 4. Overload and violation of contracted QoS

can see that running more than 3 connections resulted in QoS contract violation.

In the next experiment we analyzed system response to transient load disturbances. We fixed the number of connections, defined multiple QoS levels for each, then varied the load on the host letting the QoS-optimization algorithm run every 5 seconds to recompute the QoS levels based on the most recent estimate of γ_{effect}. Two long compilation tasks were started concurrently with packet transmission to overload the CPU. Figure 5 shows the results of a representative run. The top part of the figure shows 5 connections, labeled $C_1, ..., C_5$, where C_1 is the least important connection, and C_5 is the most important. The QoS contract for each connection had 3 QoS levels of bandwidth $1Mb$ ($M_i = 100kb, P_i = 100ms$), $0.33Mb$ ($M_i = 67kb, P_i = 200ms$) and $0.11Mb$ ($M_i = 33kb, P_i = 300ms$) respectively. Rewards were assigned proportionally to the bandwidth and weighted by connection importance. Thus, the reward for QoS level k of connection C_i was $R_i[k] = iM_i[k]$. The figure depicts the QoS level selected for each connection at every invocation of the load control module. The bottom part of the figure shows the change in the measured cost-per-packet, γ_{effect}, as well as the number of missed deadlines between successive invocations of the load control module. (A deadline miss means that M_i bits weren't transmitted within P_i time units.) As can be seen from the figure, less important connections were degraded during overload intervals to keep aggregate system throughput below saturation. The aggregate reward was thus significantly higher than in the case of indiscriminate degradation (where *all* of the connections would have been degraded below the nominal QoS level). Furthermore, it is easy to verify that the QoS-optimization algorithm performed by the load control module near-optimally utilizes the bottleneck resource. For example, consider point $t = 40$ in Figure 5. The measured value of γ is about $3.9ms/pkt$ which allows a throughput of $256pkts/second$. The aggregate throughput maintained at the chosen QoS-levels is $230pkts/second$. The utilization found by the chosen QoS-level assignment is therefore about 90%. Since the QoS level optimization algorithm runs at a slow period (5 seconds), we observed a relatively large number of deadline violations. This is because QoS level selection was not responsive to short term load fluctuations which caused some deadlines to be missed. When the experiment was repeated with the fast feedback loop enabled (with underutilization detection threshold set to 85% and overload detection set to $d = 2$, i.e., two missed deadlines). Since QoS levels were now adjusted in response to transient load disturbances, most of the deadline misses were eliminated, as shown in Figure 6. Higher values of d would result in a more lax overload detection, and thus, more missed deadlines. Lower values of the underutilization detection threshold would result in less efficient resource utilization.

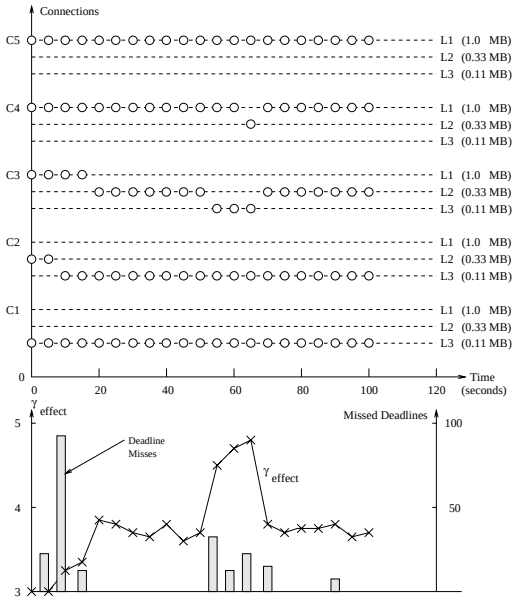

Figure 5. QoS level adaptation

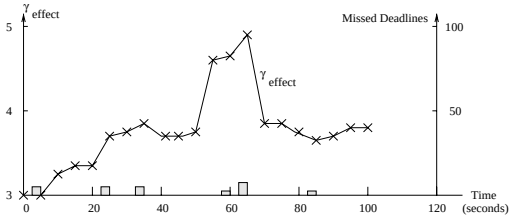

Figure 6. Improved responsiveness due to QoS level adjustment heuristic

The above experiments illustrate the function and effectiveness of the main architectural elements of our design. We highlight the fact that our design meets its

four objectives presented in the abstract. Namely, it (i) provides per-ANA guarantees on the end-host, (ii) maximizes the aggregate reward of the end-host's communication service across all clients, (iii) adapts responsively to transient overloads and resource shortage, and (iv) does not starve lower-priority service classes during the period of sustained overload (but degrades them to a lower QoS level).

7. Conclusion

We presented the design and implementation of a new model and structuring methodology for communication subsystems on end-hosts based on the notion of a flexible QoS contract. Both QoS-specification API and QoS optimization have been addressed to maximize aggregate reward based on a combination of periodic optimization and fast heuristic adjustment. We discussed monitoring and feedback mechanisms for detecting overload and underutilization conditions, as well as policies for adjusting QoS levels in response to such conditions. The impact of such mechanisms on system performance was illustrated experimentally on an actual implementation of the communication subsystem. The analysis has shown the design to be capable of meeting its stated goals.

References

[1] 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.

[2] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang. The tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on Networking*, 4(1):1–10, February 1996.

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

[4] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence. Modeling applications for adaptive qos-based resource management. In *Proceedings of the 2nd IEEE High-Assurance System Engineering Workshop*, Bethesda, Maryland, August 1997.

[5] 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.

[6] L. Georgiadis, R. Guerin, V. Peris, and R. Rajan. Efficient support of delay and rate guarantees in an Internet. In *ACM SIGCOMM*, Stanford, California, August 1996.

[7] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu. An end-to-end qos model and management architecture. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 82–89, San Francisco, California, December 1997.

[8] 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.

[9] 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.

[10] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1044–1056, Oct. 1994.

[11] V. G. Kulkarni and N. Gautam. Leaky buckets : Sizing and admission control. In *Proceedings of the 35th IEEE Conference on Decision and Control*, Kobe, Japan, 1996.

[12] 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.

[13] A. Mehra, A. Indiresan, and K. G. Shin. Resource management for real-time communication: Making theory meet practice. In *2nd IEEE Real-Time Technology and Applications Symposium*, June 1996.

[14] 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.

[15] 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*, pages 90–99, May 1994.

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

[17] 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.

[18] H. Schulzrinne. RTP: The real-time transport protocol. In *MCNC 2nd Packet Video Workshop*, volume 2, Research Triangle Park, North Carolina, December 1992.

[19] H. Schulzrinne. A comprehensive multimedia control architecture for the Internet. In *NOSSDAV*, St. Louis, Missouri, May 1997.

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

[21] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, September 1993.