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

The proliferation of communication-intensive real-time applications with "elastic" real-time constraints, such as streaming stored video, and the advent of differential services over the Internet require a new design for end-host communication subsystems. The design should (i) provide per-flow or per-service-class guarantees on the end-host, (ii) maximize the aggregate utility (or revenue) of the end-host's communication service across all clients, (iii) gracefully adapt to transient overload and resource shortage, and (iv) not starve lower-priority service classes during the period of sustained overload. This paper presents a new architecture for QoS-adaptive end-host communication subsystems in order to meet the above-stated goals. This architecture differs from others in that it provides *generic* end-host communication subsystem support for QoS-adaptive communication, as opposed to devising QoS adaptation for a *particular* type of real-time applications.

Our architecture provides each client its contracted QoS, while adapting gracefully to transient overload and resource shortage. This is done by allowing clients to specify *multiple* acceptable levels of service (*QoS levels* for short), and express their relative desirability by assigning them the corresponding "rewards." The client-specified QoS levels express a nominal mode of operation as well as degraded modes. Rewards may be tied to certain application-level performance measures, or may represent actual monetary payment for receiving service at the corresponding QoS level. Specifying multiple acceptable QoS levels for each client allows the server to *predictably* degrade client's QoS under transient overload or resource shortage. This, in turn, leads to a more flexible, yet predictable *QoS contract* between the server and clients. The proposed architecture is geared primarily towards soft real-time applications, but, in the presence of resource-reservation support, it can provide hard real-time guarantees. We implemented the architecture on a network of Pentium-based PCs each running under OG MK7.2. The evaluation results demonstrate 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 the National Science Foundation under grant No. MIP-9203895. 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 design for soft QoS-adaptive real-time applications. Adaptation is exploited to improve overload performance. In a system where resources (such as CPU cycles) cannot be preallocated to protocol processing, or where the underlying OS does not support real-time scheduling and admission control, overloading the CPU may significantly degrade the communication subsystem. Optimal message-scheduling algorithms, such as EDF, will perform poorly under overload. On the other hand, fixed-priority scheduling algorithms will starve low-priority connections. Commonly-used FIFO queuing of messages will introduce unpredictable delays. Instead, we propose a communication subsystem architecture that "optimizes" real-time performance in the presence of overload. The architecture provides a new, flexible notion of QoS contract between the communication subsystem and its clients that expresses both the nominal and overload QoS requirements. We present the design and implementation of a resource-management mechanism that enforces the proposed notion of QoS contract.

Our work is complementary to the research on real-time communication protocols [1–4], network support for QoS [5, 6], and real-time operating system support [7–10]. We focus on the communication subsystem within an end-host, and develop an end-host communication resource-management architecture for QoS guarantees that is independent of the particular communication protocol stack or specifics of the client application. Different communication architectures have been proposed to support QoS guarantees. Examples include the QoS-A framework [11], the Heidelberg QoS model [12], V-net [13], NetWorld [14], COMETS' Extended Integrated Reference Model (XRM) [15], the OMEGA end-point architecture [16], and the QoS Broker [17]. A good survey of these and other communication architectures can be found in [18]. Such architectures are designed specifically for multimedia applications. By contrast, we present a generic communication subsystem architecture for end-hosts that is capable of providing hard guarantees if the OS and network support resource reservation, yet is flexible enough to adapt to variations in resource capacity when no reservation support is present. Real-time communication protocols such as RTP [3], can be used with our scheme to help clients adapt application-level performance to network delays. RSVP [4], or real-time channels [1, 2] can be used to reserve host and network bandwidth, when applicable. Unlike "hard real-time" communication architectures [19], our design makes no assumptions about the existence of resource-reservation support both on the end-host OS and in the network, although it can make use of such support if available. Since soft real-time applications often run on non-real-time platforms (e.g., a video-phone on a PC or Sun workstation), we do not require exact system-load characterization and profiling. Instead, a self-tuning mechanism is incorporated into our architecture to adjust to the measured load and resource conditions.

The main contribution of our architecture is its support for a more flexible form of QoS contracts, that is especially suitable for adaptive soft real-time applications. 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 acceptable degraded 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. Under overload, an optimization algorithm is invoked to determine how to degrade individual clients. Thus, instead of inflicting arbitrary degradation, the aggregate system bandwidth is subdivided among clients in a controlled fashion that optimizes some performance measure. If the overload is severe, however, it may be impossible to provide the service even at the lowest QoS level for each client. Thus, each QoS contract specifies a *QoS-violation penalty*. Resource management ensures that clients will either receive their contracted service (and get "billed" for the QoS level), or receive a "rebate" corresponding to the specified QoS-violation penalty. The goal of the server is to maximize its reward. Thus, the optimization algorithm chooses clients' QoS levels that optimize the aggregate reward subject to resource constraints, given the information on rewards and QoS-violation penalties. Rewards can be seen as an abstract representation of an arbitrary performance measure. In applications where rewards and penalties represent actual monetary values, it may be more appropriate for the service provider, not clients, to set them. The optimization will then maximize aggregate revenue. In other applications the reward of a QoS level may represent the achieved value of some physical measure for the client, and the rebate is the client-perceived cost of QoS violation. An example of how rewards and penalties can be set in such a case is described in [20]. Either way, the resource-management mechanism for enforcing clients' QoS contracts and maximizing the aggregate reward remains the same. Hence 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 disturbances and resource shortage. This is in sharp contrast to indiscriminate degradation that applies to premium-service clients as well as basic-service clients alike.
- Not starve lower-priority service classes under prolonged overload conditions. Such starvation has often been observed in static-priority scheduling schemes.

In Section 2 we define the proposed new abstraction and present the corresponding communication subsystem architecture. Two main components of the architecture are elaborated on in Sections 3 and 4, respectively. Section 5 provides details of 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. The semantics of a QoS level are described in Section 2.1. Section 2.2 presents the proposed notion of the QoS-contract based on expressing multiple acceptable QoS levels. The issue of quantifying the relative desirability (reward) of these QoS levels by the client is elaborated on in Section 2.3. Finally, the general architecture of the communication subsystem is highlighted in Section 2.4.

2.1 QoS Levels and Communication-Capacity Reserves

In order to provide a predictable communication service, the amount of communication traffic generated at the source must be bounded. The leaky bucket model [21] 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 view-point, 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 parameter B_i. The reserve can be of *sender* or *receiver* type. At the sender side, the reserve holds enough computing resources for M_i bytes to be transmitted every P_i units of time. The buffer 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 $N_{max}(t)$. Bytes generated in excess of the above are said to be *non-conformant*. Guarantees are given to conformant bytes only. Since M_i conformant bytes are guaranteed to be transmitted every P_i units of time and since the maximum number of conformant bytes awaiting transmission is $B_i M_i$, a conformant byte is guaranteed to be transmitted within $B_i P_i$ time units after its generation at the source. Thus, if it is generated at time t, its transmission deadline is $D_i = t + B_i P_i$. Non-conformant bytes may be dropped or delayed until they become conformant in which case they will be transmitted within $B_i P_i$ time units after their conformance time. 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. Bytes in excess of M_i received within a period P_i are non-conformant. The buffer size parameter B_i specifies the maximum size of the delivery buffer 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

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

by the protocol stack at the receiver and deposited in a delivery buffer. In a particular implementation, the application may be signaled, upcalled, or otherwise notified of data delivery. The communication service guarantees a conformant data byte to be deposited into the delivery buffer no later than P_i time units after it has been received. The application must read (i.e., drain) the delivery buffer within $B_i P_i$ time units of the packet's reception or else the buffer may overflow.

Note that the QoS level (or a communication-capacity reserve) as defined above is an end-host level abstraction, and not an end-to-end connection-level abstraction. For example, it can be used with connectionless protocols such as UDP to reserve enough computing capacity to handle a given amount of traffic at a communication endpoint. In connection-oriented protocols, it will be the responsibility of the signaling phase to set aside appropriate communication-capacity reserves on both the sender and the receiver. The communication-capacity reserve abstraction is also ambiguous regarding the type of network used. In particular, it does not imply that network bandwidth is actually reserved for a connection, since the network might not support per-flow bandwidth reservation. We decouple the concerns of end-host resource reservation from those of network resource reservation. The latter concern should be addressed at the network communication protocol layer, while the former is a concern of end-host resource management.

2.2 The QoS Contract

We are interested in supporting applications with potential for QoS adaptation to variations in available resource capacity. Examples include (i) a video source capable of decreasing image resolution, color depth, or frame rate when informed by the underlying transport medium of the lack of available resources, (ii) participants in a distributed real-time protocol that may tolerate a larger end-to-end delay if a given smaller delay cannot be guaranteed due to overload, and (iii) automatic control applications which may adapt sampling intervals at the expense of reducing loop stability margins, to name a few. In other words, applications' rate, data size, and burst constraints are generally flexible (i.e., fall within a *range* of allowable values). The QoS specification API should allow such applications to express their flexibility with regard to acceptable QoS. Specifying the minimum requirements is the simplest approach. However, it does not provide clues as to how available resources should to allocated in excess of the minimum requirements. A more expressive API is to specify both the maximum (or nominal) and minimum requirements. However, it has several disadvantages:

- In general, the application may not be able to adapt itself to changes in received communication QoS within a continuous range between its specified minimum and maximum requirements. Instead, it may only have discrete options such as sending 16 color video, 4 color video, or black & white video. Thus, for example, receiving enough resources to transmit an 8 color image may be pointless.
- Degradation of client-perceived utility may not be linear in the received QoS level. For example, while the user may not perceive a significant difference between a 16 color and a 4 color image, a

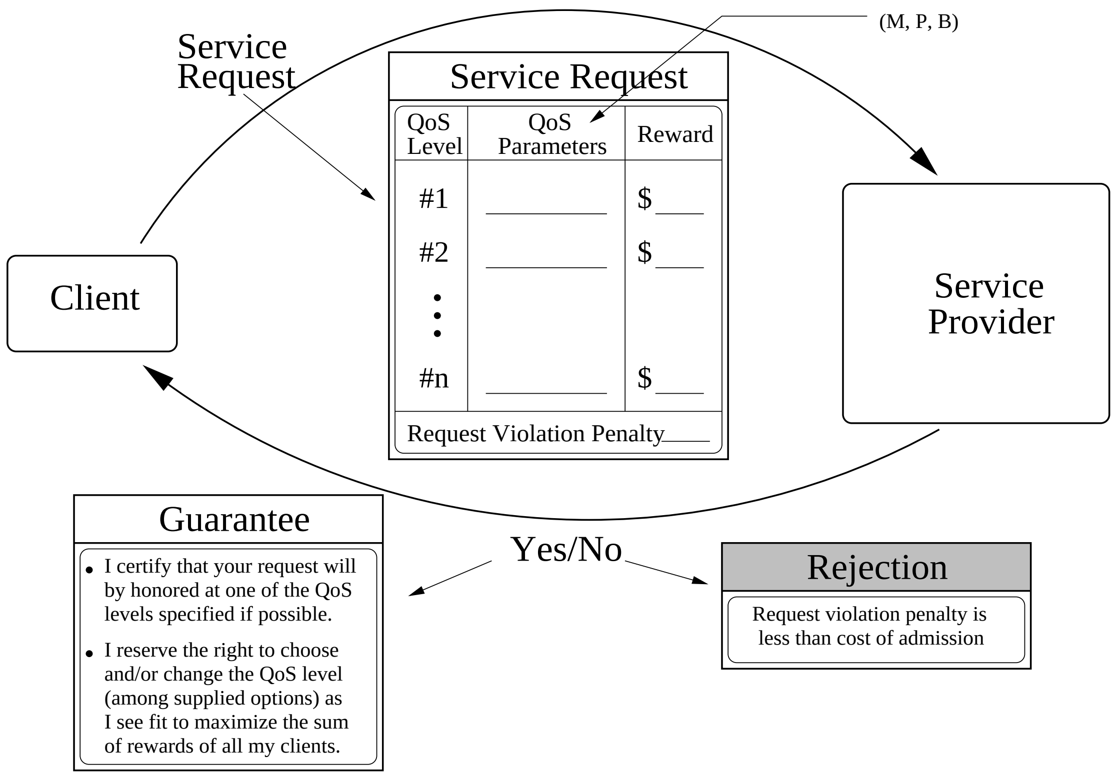

Figure 1: Connection establishment request

B&W image may be highly undesirable.

The suggested API is, therefore, based on multiple QoS levels. The QoS contract Q_i specifies alternative *QoS levels*, 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. All rewards must have the same units across clients. For example, in our implementation, clients are restricted to specify rewards in a range from 0 to 100%, which is then scaled within the communication server by the importance level of the client via a privileged interface. In another implementation, clients may be given "credit lines" in accordance with their importance, then allowed to specify reward values that are debited to their capital. The QoS-violation penalty is the cost incurred if none of the requested alternative QoS levels can be provided by the communication subsystem. Figure 1 shows an example connection-establishment request.

Rewards and violation penalty are not a redundant expression of the same concept. They quantify different aspects of client preferences. For example, a multimedia user may attach a very high violation penalty to an important video-lecture while assigning low rewards to the resolution of video transmission during the lecture if he/she is more interested in the audio contents. The same user may attach a relatively low violation penalty on viewing Jurrasic Park while assigning sufficiently high rewards to the resolution of the movie's transmission. This tells the system that in case of overload the resolution of the lecture should be degraded first, before that of Jurrasic park, but if resources are no longer sufficient to maintain both connections, Jurrasic Park should be the one to be terminated. Note, also, that specifying only one acceptable QoS level with a default (e.g., infinite) violation penalty reduces this mechanism to traditional on-line admission control.

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. This gives the subsystem enough leeway for QoS adaptation and the optimization of aggregate reward.

2.3 Reward Estimation

The problem of specifying rewards for different QoS levels (as well as the QoS levels themselves) warrants additional discussion. In real-time applications interacting with a human user (rather than a physical environment) QoS levels and rewards are essentially 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 information may be set by the end users at the time of service request 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. Furthermore, an "expert" user interface may allow users to customize their applications by manipulating the preset definition of "poor", "good" and "excellent" QoS. This is somewhat similar, for example, to adjusting the brightness and contrast of a regular TV screen, or adjusting the multiple bands of a stereo equalizer for a certain playback quality. In a system deployed for commercial purposes, 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 an objective performance measure. For example, in [20] 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 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 elastic service requirements, whenever such information is available.

2.4 Communication Subsystem Architecture

In this subsection we highlight 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. In order to do so, each client (or service class) is handled within the communication server by a *separately schedulable* entity. In the rest of this paper we 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 to the communication server and is scheduled in accordance with the client's assigned QoS level. Having different ANAs for different clients allows serving clients concurrently at different QoS levels.

The second objective of our architecture is to maximize server's aggregate reward. This is accomplished by a proper choice of QoS levels that "optimally" utilizes the 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 aggregate achieved reward subject to resource constraints our design utilizes a combination of (i) a reward optimization algorithm implemented by a load control module that uses a mathematical estimate of current load for admission control and QoS-level selection, and (ii) a monitoring module that uses actual measurements to compute the parameters of the mathematical load model. 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 reward 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 require immediate response. Since re-executing the reward optimization algorithm upon every load change may consume too much bandwidth, we provide a fast feedback loop, implemented by the moniroing 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, which satisfies the fourth objective of the architecture. Figure 2 depicts the major components of 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. The component is labeled *load control* in Figure 2. 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

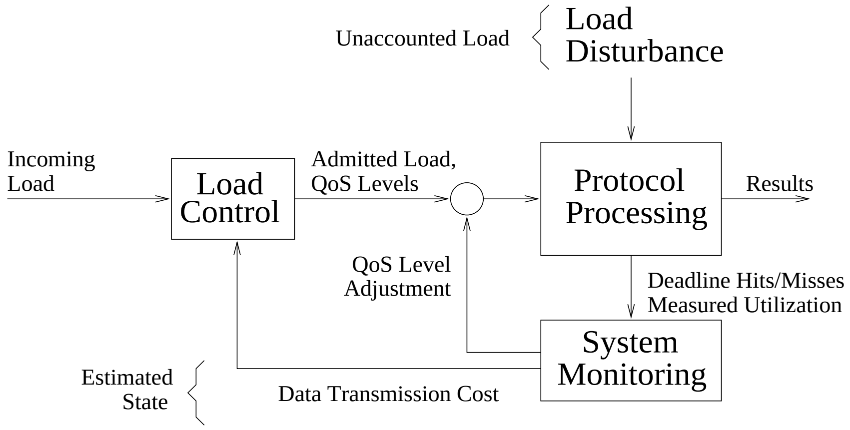

Figure 2: The Communication Subsystem

to the case where these parameters vary dynamically. Assume that the communication subsystem is allotted an amount U_{server} of CPU utilization, and that the host has an amount BW of network bandwidth dedicated to it. This may be the case, for example, with point-to-point networks, ATM or FDDI. The load control module deals with ANAs as the objects to be scheduled, and assigns each a QoS level among those specified in its QoS contract. Traffic handled by an ANA is policed by the ANA itself to conform to the QoS-level specification. It is therefore transparent to the load control module.

3.1 QoS-Level Selection

Upon invocation, the load control module executes a QoS-level selection algorithm. QoS levels must be selected to maximize the aggregate reward. This is an NP-complete problem because it is reducible to the multiple-choice knapsack problem. In this section we present a near optimal pseudo-polynomial-time algorithm for QoS-level selection performed by the load control module. It discretizes the solution space to reduce the NP-complete problem to one of complexity $O(nL_{av})$ where n is the number of established ANAs and L_{av} is the average number of QoS levels defined for an ANA. Note that this is the same complexity as reading problem input (all possible QoS levels of all ANAs).

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$_i$, have 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 *rejection level* (at which the client receives no service), with the cost of QoS violation as specified by 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., $R_i[k] = -V_i$). Let $f_c(x)$ be the cost function which gives the total time it takes to process and transmit x bytes. Let us define $E_i[k]$ such that:

$$E_i[k] = f_c(M_i[k]). \tag{3.1}$$

In general, f_c is nonlinear. The cost is not proportional to the number of bytes transmitted due to cache effects, packetization, and depacketization overheads. An accurate expression of $f_c(x)$ is given in [22]. The processor utilization requested by a QoS level $L_i[k]$ is:

$$U_i[k] = E_i[k]/P_i[k]. \tag{3.2}$$

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. To reduce the NP-complete 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 us define $S(i,U)$ to be the problem 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 problems $\{S(i,\delta), S(i,2\delta), ..., S(i,U_{server})\}$. Given n ANAs in the subsystem we need to solve the problem $S(n,U_{server})$. The following recursive relation exists:

$$S(i,U) = \max_{1 \leq k \leq m_i} \{R_i[k] + S(i-1, U - U_i[k]) | BW_i[k] + BW(i-1, U - U_i[k]) \leq BW\} \tag{3.3}$$

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

$$BW(i,U) = BW_i[k^*] + BW(i-1, U - U_i[k^*]) \tag{3.4}$$

For the special case of $i = 1$,

$$S(1,U) = \max_{1 \leq k \leq m_1} \{R_1[k] | U_1[k] \leq U, BW_1[k] \leq BW\} \tag{3.5}$$

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

$$BW(1,U) = B_1[k^*] \tag{3.6}$$

The above recursive relation computes the solution to a problem in $S(i,*)$ given solutions to the problems 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

$i = 1, ..., n$. Thus, there is a total of only nK problems $S(i, U)$ to be solved. Solving all of these problems (in increasing order of index i) will therefore take $O(K \sum_{1 \le i \le n} m_i)$ which is equivalent to $O(K n L_{av})$, 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, which makes the complexity of the algorithm $O(n L_{av})$. 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 3.3
4     compute BW(i,U) from Equation 3.4
5 return S(n,U_server)
```

The above algorithm exhaustively searches the space of all solutions in the descretized CPU utilization space and returns the "optimal" one. The returned solution assigns a QoS level for each ANA such that aggregate server reward is maximized. From the 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. Note that since ANAs handle only communication traffic, their CPU requirements are roughly proportional to their communication bandwidth requirements. Thus, it suffices to search the utilization space of only one of the two resources (CPU, or communication bandwidth). The recursive Equations 3.3, and 3.4 show that the communication bandwidth constraint is taken care of simply by pruning parts of the search space that violate the constraint. In the worst case (e.g., when $U_{server} = 0$, or $BW = 0$), the solution returned by the above algorithm will specify the rejection level (defined earlier) for all ANA. For the special case of admission control an incremental form of the algorithm is derived next to amortize its computation cost on several ANAs admission requests.

Note from **Algorithm A** that when a request for creating ANA$_n$ arrives, the sets of problems $S(1, *)$, ..., $S(n-1, *)$ will have already been computed at admission time of ANA$_{n-1}$. Thus, at admission time of ANA$_n$, we need to compute only the incremental set of problems $S(n, *)$, which is an $O(L_{av})$ computation. Furthermore, of all problems in set $S(n, *)$, only $S(n, U_{server})$ is in the "critical path" of admitting the new request. The other problems in $S(n, *)$ can be computed later in the background to set the stage for admitting a new ANA$_{n+1}$, when the new admission request arrives. The incremental admission-control algorithm executed upon request arrival is thus as follows:

```
admission_control(request ANA_n)
1 compute S(n, U_server) from Equation 3.3
2 if S(n, U_server) < S(n-1, U_server) - V_n reject ANA_n
3 else accept ANA_n
4 compute remaining problems in set S(n, *)
```

3.2 Practical Approximation of QoS-Level Selection Algorithm

The algorithm, as presented above, is a multiple choice knapsack problem with the CPU being the knapsack, and network bandwidth requirements being an additional constraint. Alternatively, it could have been coded with network bandwidth being the knapsack and CPU requirements being the additional constraint. Since CPU requirements are roughly proportional to communication bandwidth requirements, it can be shown that if the knapsack problem is solved for the *bottleneck* resource, the additional constraint will always be satisfied. For better efficiency of packet handling, 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. As an approximation, we lump the components of $f_c(x)$ into a per-packet processing cost. Thus, we use the 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 3.2, and 3.1, the bottleneck resource utilization requested by a QoS level $L_i[k]$ is:

$$U_i^{bottle}[k] = E_i[k]/P_i[k] = \gamma\lceil M_i[k]/MTU \rceil / P_i[k]. \tag{3.7}$$

Furthermore, Equations 3.3 and 3.5 for the bottleneck resource can be simplified by omitting the always-satisfied constraint on the other resource:

$$S(i, U) = \max_{1 \le k \le m_i} \{R_i[k] + S(i-1, U - U_i^{bottle}[k])\} \tag{3.8}$$

$$S(1, U) = \max_{1 \le k \le m_1} \{R_1[k]\} \tag{3.9}$$

Finally, let us examine γ, the per-packet processing cost of the bottleneck stage. Such cost is usually difficult to measure, since it requires accurate profiling of the system. This is complicated by the fact that resources (e.g., CPU or network bandwidth) may not be entirely available to the communication subsystem. For example, the CPU may be shared with other tasks executing on the host, and network bandwidth may be shared with other hosts on the same network. Let r be the resource used by the bottleneck stage and λ be the fraction of that resource's bandwidth available to the communication subsystem. Unless λ is fixed using such means as processor capacity reserves or WFQ (weighted fair queuing), it may be difficult to measure. Without loss of generality, let us scale both resource capacity and demand by an identical factor $1/\lambda$. The available fraction of the bottleneck resource bandwidth becomes λ/λ, i.e., the entire resource. Furthermore, the effective processing cost γ_{effect} is scaled so that $\gamma_{effect} = \gamma/\lambda$. Although both γ and λ are difficult to measure, their ratio γ_{effect} is simply the inverse of system throughput (in packets/unit time). Intuitively, if the entire bottleneck resource was dedicated to communication packet processing, then the cost of processing a packet is the the time elapsed, divided by the number of packets processed. Consequently, **Algorithm A** can be simplified as follows:

```
Algorithm A'
1 for i = 1 to n
2   for U = 0 to 100% in steps of δ
3     compute S(i, U) from Equation 3.8
4 return S(n, 100)
```

where utilization values in Equation 3.8 are computed from:

$$U_i[k] = E_i[k]/P_i[k] = \gamma_{effect}\lceil M_i[k]/MTU\rceil/P_i[k]. \tag{3.10}$$

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, 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. Since the algorithm incurs a relatively high cost, it is more suitable for adaptation to long-term resource variations. A different mechanism is needed to adapt to short-term load disturbances. We integrate both mechanisms into one load monitoring and feedback architecture as described in Section 4.

3.3 Client Aggregation

The complexity of the algorithm presented in the previous subsection has a multiplicative constant that grows with the resolution, δ, of possible utilization values. Higher resolutions (i.e., smaller δ) will allow expressing finer utilization values at the expense of increased computation. Due to discertization of utilization values, each admitted ANA will increase utilization by at least δ, meaning that the maximum number of admitted ANAs is upper-bounded by $100/\delta$. This may be acceptable for an individual user workstation running a limited number of real-time communication-bound applications (e.g., a few video-on-demand connections). However, the bound poses a serious limitation on the design

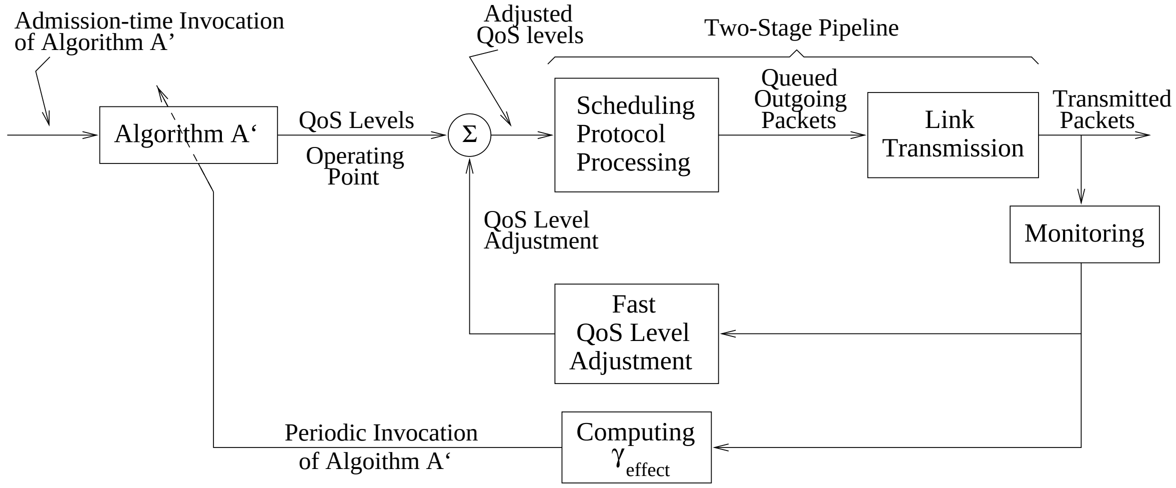

Figure 3: Monitoring and feedback architecture

of scalable communication servers for multiple users. Fortunately, it is possible to aggregate clients of such servers into a number of generic classes. ANAs can then be requested for a class. 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 preselected 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 surplus clients in the class. Since the number of classes is typically much smaller than the number of clients, the value of δ need not be very small to give an adequate resolution for per-class resource reservation.

4 Monitoring and Feedback Control

In the previous section we described how QoS levels are near-optimally selected for ANAs. However, the amount of resources available to the communication subsystem often varies dynamically in an uncontrolled manner, for example, due to network congestion or sharing the host among several tasks. Instead of trying to model the exact behavior of these fluctuations or alternatively use a pessimistic worst-case analysis, our philosophy is to (i) select the QoS levels of ANAs based on approximate information (γ_{effect}), then (ii) adjust the selected QoS levels via a fast monitoring and feedback loop if overload or underutilization is detected. These two techniques are elaborated upon in more detail in the following two subsections. Figure 3 depicts a more detailed view of this architecture.

4.1 Computing γ_{effect}

The maximum system throughput V (maximum number of packets transmitted) is dynamically estimated and its inverse $\gamma_{effect} = 1/V$ is computed by the monitoring module to determine the per-packet processing cost. Every period, P, the monitoring module looks up (i) the number of packets, N, transmitted within P, and (ii) the total time μ, out of P, during which some ANA(s) was/were ready or running. The ratio μ/P represents system utilization by scheduled ANAs (taking into account the tasks that preempt their execution). The maximum system throughput is approximately $V = N/\mu$, and $\gamma_{effect} = \mu/N$. To prevent frequent fluctuations in γ_{effect}, its value is averaged over W successive measurements, thus filtering out high frequency variations. The average γ_{effect} captures long-term load variations. The QoS-level assignment algorithm is invoked with a period WP to recompute the QoS levels of all ANAs accordingly. The computed QoS levels may be viewed as the "operating point" of the system. For the current implementation, we use $P = 100ms$, and $W = 50$.

4.2 Fast Feedback Loop

The adaptation mechanism mentioned above is insensitive to short-term transient load disturbances which tend to be averaged out. 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 transmitted packets are meeting their transmission deadlines D_i, as defined in Section 2.1. When d deadlines are violated, "overload" is flagged. The parameter d can be configured according to deadline criticality. Underutilization conditions are detected by inspecting the effective utilization μ/P. When it drops below a preconfigured threshold *and* some of the ANAs are scheduled below their maximum QoS level, "underutilization" is flagged. 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:

QoS Level Adjustment Heuristic

if *overload* then
  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)
  Degrade C_i to next QoS level.
if *underutilization* then
  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)
  Promote C_i to next QoS level.

The algorithm will alter system load potentially 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 Contract Enforcement

The communication subsystem must ensure that each ANA receives service at its assigned QoS level. For example, if the selected QoS level for a sender (receiver) ANA_i is $L_i[k]$, then the communication subsystem must ensure that (i) $M_i[k]$ bytes can be sent (received) each period $P_i[k]$, and (ii) there is enough buffer space to store $M_i[k]B_i[k]$ bytes generated (delivered) for the reserve. While the latter requirement is satisfied by adequate memory allocation, the former is an end-host scheduling requirement. A proper scheduling mechanism must thus be implemented to ensure QoS contract enforcement.

A thread-per-connection model has been suggested for communication subsystem design in [19]. The model facilitates per-connection policing to prevent misbehaving communication clients from violating their traffic specification, and makes it possible to assign thread priorities to ensure the required QoS on a per-connection basis. We extend this model to ANAs, offering a thread-per-ANA architecture. We implement a user-level thread package whose scheduler transparently executes the near-optimal QoS level selection 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. Although the package is illustrated in the context of scheduling the communication subsystem, its design was intended for use in a broader context. It has been implemented with the intention of supporting generic adaptive server architectures. For the particular case of a communication server, a generic ANA thread was written to perform protocol processing, packetization/depacketization, data buffering, policing, and client aggregation.

5.1 $q$threads: A Thread Package for Adaptive Server Design

Our implementation of the communication subsystem uses $q$threads — a thread package we developed for adaptive server design. 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 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}. Apart from the CPU requirement, each QoS level may be associated with arbitrary passive resource requirements specified by a resource requirement vector $(R_{i,1}[k], \cdots, R_{i,r}[k])$. $R_{i,j}[k]$ specifies the amount of the jth resource to be allotted for the execution of thread T_i at QoS level k. When a thread is created using the *qthread_create()* primitive, a thread admission-control algorithm is invoked to compute a QoS level for each thread so that the aggregate achieved reward is maximized. This algorithm has been described in Section 3. The *qthread_create()* call fails if the admission-control 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. Each ANA is implemented as a separate $q$thread.

5.2 Implementing Adaptive Negotiation Agents

In this subsection we describe our implementation of ANAs using $q$threads. The architecture is implemented as a communication library called *CLIPS* (Communication Library for Implementing Priority Semantics). It offers a generic adaptive communication resource-management mechanism on end-hosts which is independent of the specific communication protocol stack used. The main tasks implemented 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 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. We gain such flexibility by using x-kernel for protocol development. x-kernel exports a uniform protocol interface and allows configuring arbitrary protocol stacks.

- *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 the ANA once it has used its allotted packet budget for the given period, until the start of the next period.

- *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 levels selected by $q$threads. Outgoing packets are therefore queued at the network device interface in a priority heap sorted by handler 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.

Figure 4 depicts the architecture on the sender side. It shows the modular separation between the generic $q$thread scheduling package and the communication support offered by *CLIPS*. It also shows the separation of *CLIPS* communication resource-management mechanism itself from the particular communication protocol stack used. The receiver side is similar except that data is demultiplexed and delivered to the application.

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 (OG).[2] MK 7.2 is a derivative of CMU Real-Time Mach. The communication subsystem architecture was implemented using the facilities of OG's CORDS environment which are based on x-kernel support originally developed at the University of Arizona. The advantage of using x-kernel in our platform lies in its uniform protocol interface that allows us to compose arbitrary protocol stacks in a flexible manner. An x-kernel protocol stack can be configured from protocol objects by specifying a protocol graph of the communication subsystem. The specification is then compiled into the desired protocol stack. This flexibility enables the separation between the protocol stack and the resource-management mechanism. CORDS adds to the x-kernel

[2]OG is formerly known as the Open Software Foundation.

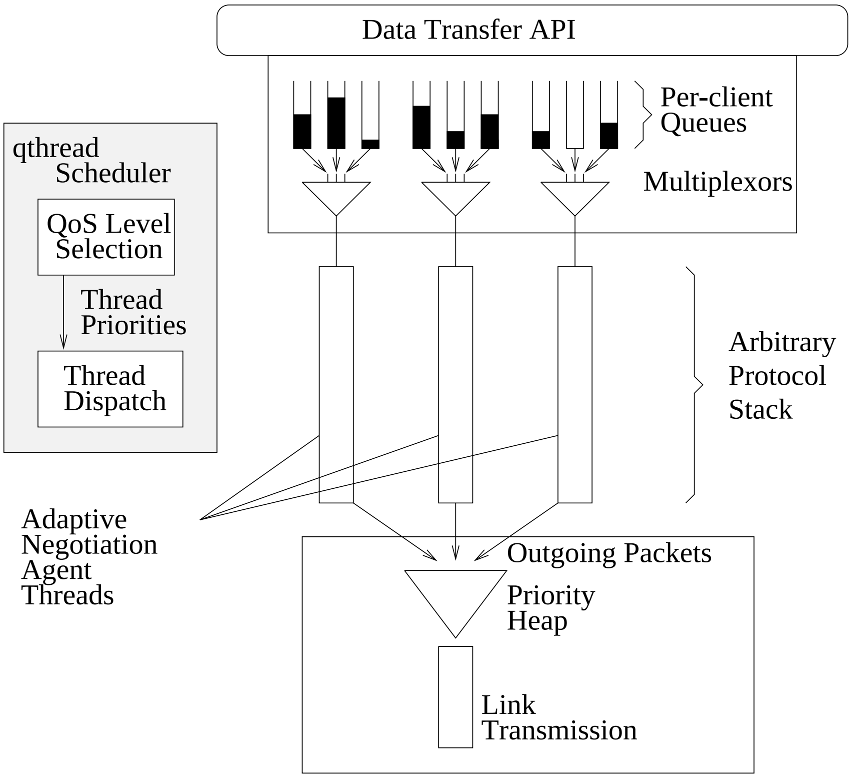

Figure 4: CLIPS Communication Architecture

framework the notion of per-connection passive resource reservation. It exports the abstraction of *paths* which can be associated with particular end-points, or end-to-end connections. The framework allows charging the consumption of resources such as memory bandwidth and message buffers at different layers of the protocol stack to the same path, which facilitates the implementation of per-ANA resource management. It also implements per-path input thread pools to shepherd incoming traffic. In the CORDS framework, the identity of a path is encoded in each packet's header. This identity is used for early demultiplexing of arrived packets at the receiver onto the right input packet buffer, as well as for invoking a shepherd thread from the right input pool. The framework allows differential services by assigning different priorities to threads of different paths. We changed CORDS to use threads whose priority is automatically set by our communication subsystem scheduler in accordance with the current QoS-level selection.

In the present implementation, the communication subsystem is realized as a CORDS server running on top of MK 7.2 microkernel. The CORDS framework provided on MK 7.2 gives an easy way of migrating CORDS server code into the kernel by setting appropriate kernel compile-time flags. Thus, the communication server can be developed more effectively in user space, then migrated into the kernel for efficiency if desired. An additional advantage of using the CORDS framework is its availability on widely-used platforms such as Windows NT. Figure 5 shows the end-host architecture, especially the communication server implementing the communication protocol stack. It also shows the interface between the application(s) and the server in our implementation environment.

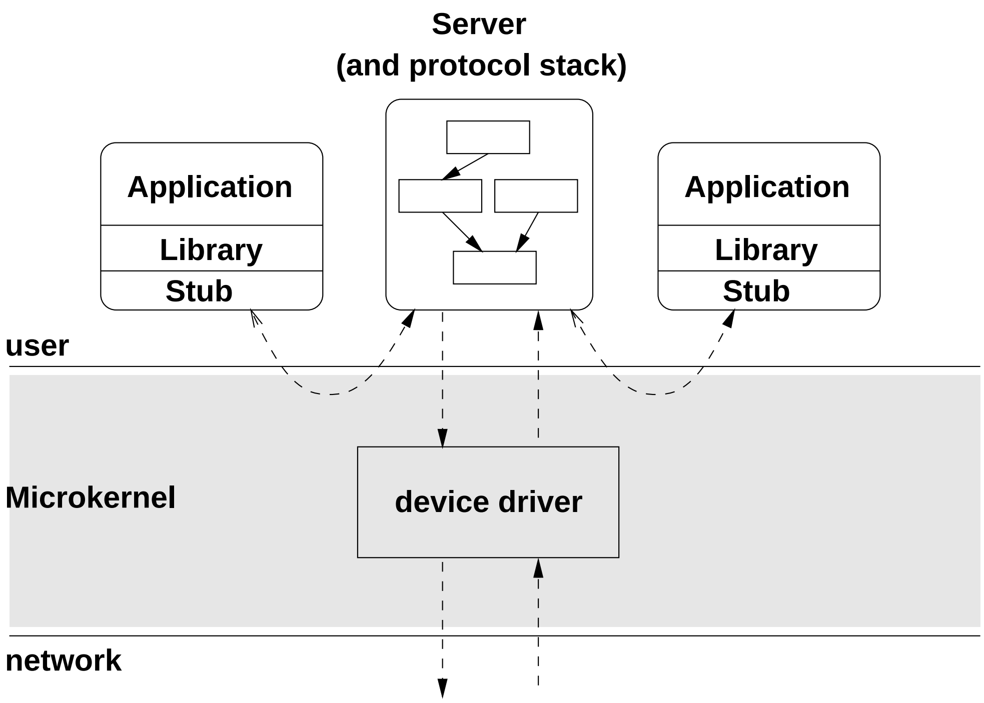

Figure 5: Communication architecture

6 Evaluation

We conducted several experiments to stress-test the performance of the implemented communication subsystem server and verify its ability to meet the contracted QoS requirements. To generate traffic, 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 hence call it a *generator thread*. An ANA was created for each connection to enforce the QoS contract, and police the generator. Figure 6 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.[3] 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. In particular, when a generator thread fills up the input buffer of its ANA, the ANA blocks the source until some messages are dequeued from the buffer. Since the ANA dequeues messages from the buffer at the rate specified by the current QoS level, it implements a leaky-bucket-like policing mechanism that meets the maximum rate constraint.

The rate-policing mechanism alone, however, is not sufficient for enforcing the QoS contract. In Figure 7 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

[3]Where B, in this section, refers to $Bits$.

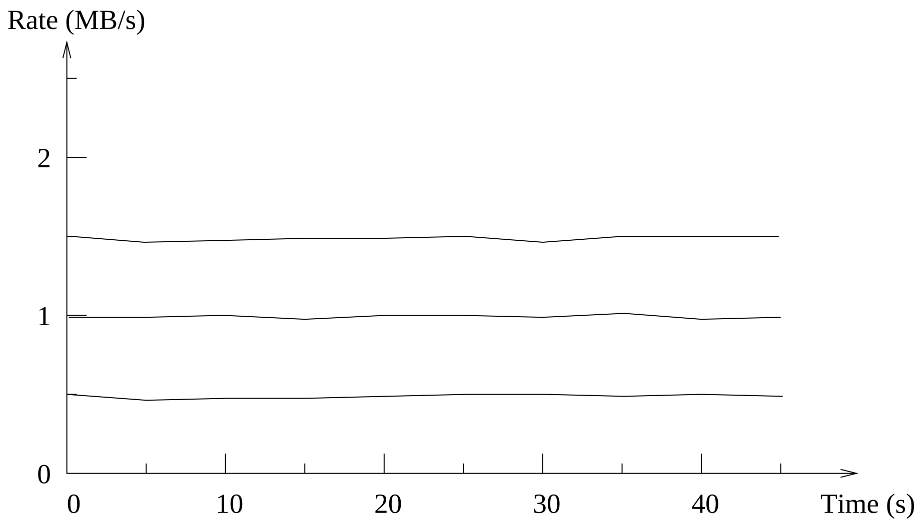

Figure 6: Policing effect

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 $3.7MB$. 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. No *a priori* system profiling was necessary. 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 7 we can see that running more than 3 connections results in QoS contract violation, indeed.

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 optimization algorithm implemented by the load control module run periodically (every 5 seconds) to recompute the QoS levels based on the most recent estimate of $\gamma_{effective}$. Two long compilation tasks were started concurrently with packet transmission to overload the CPU. Figure 8 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, $\gamma_{effective}$, as well as the number of missed deadlines between successive invocations of the load

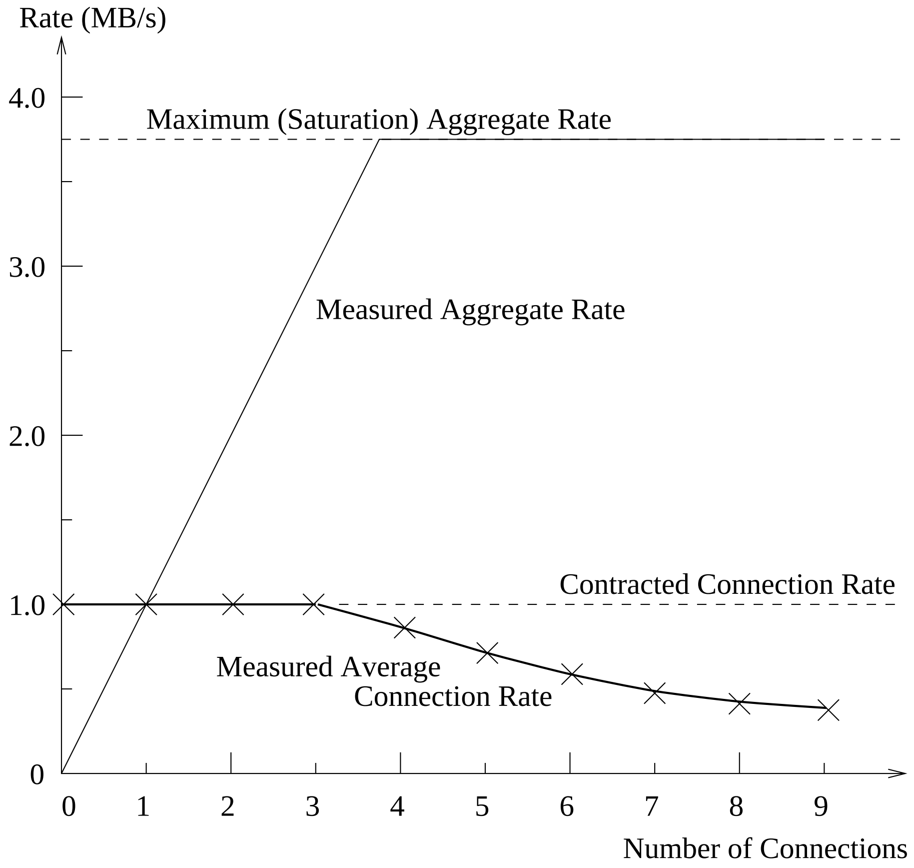

Figure 7: Overload and violation of contracted QoS

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. 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 to change the QoS levels in response to transient load disturbances, most of the deadline misses were eliminated, as shown in Figure 9.

Although the fast feedback loop is more responsive, it cannot replace the near-optimal algorithm. This is because the fast feedback loop adjusts QoS levels based on an approximate heuristic. When the heuristic is invoked near a global optimum (computed by the optimization algorithm) the hill-climbing adjustment in QoS levels is likely to find that optimum or a nearby value. However, if the heuristic is used on its own to determine QoS levels, it is almost certain to get trapped in a local maximum that underutilizes the system.

The above-presented experiments illustrate the function and effectiveness of the main architectural elements of our design. For space limitations, we do not present, in the paper, a complete analysis of

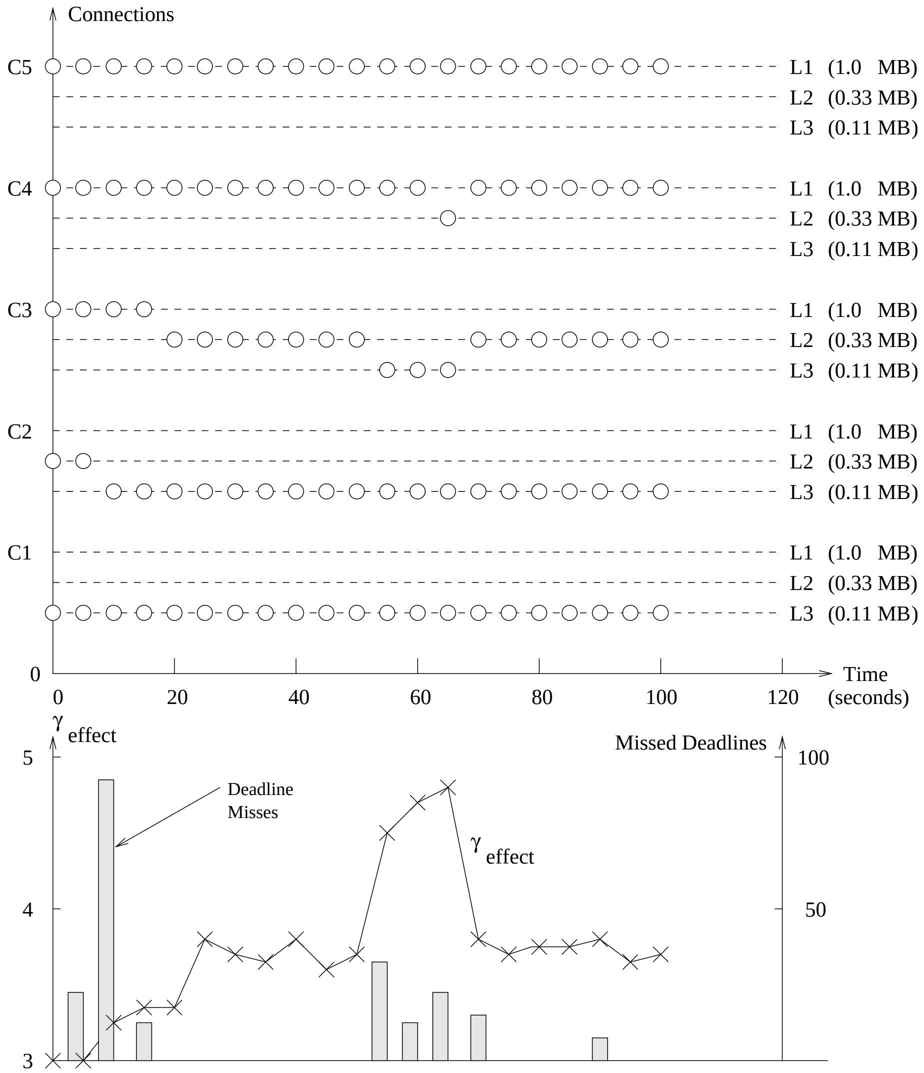

Figure 8: QoS level adaptation

system performance. We merely 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 Conclusions

We presented a new abstraction and structuring methodology for communication subsystems on end-hosts based on a more flexible notion of a QoS contract. We elaborated on the QoS specification API which allows the application developer to define flexible communication requirements by specifying a

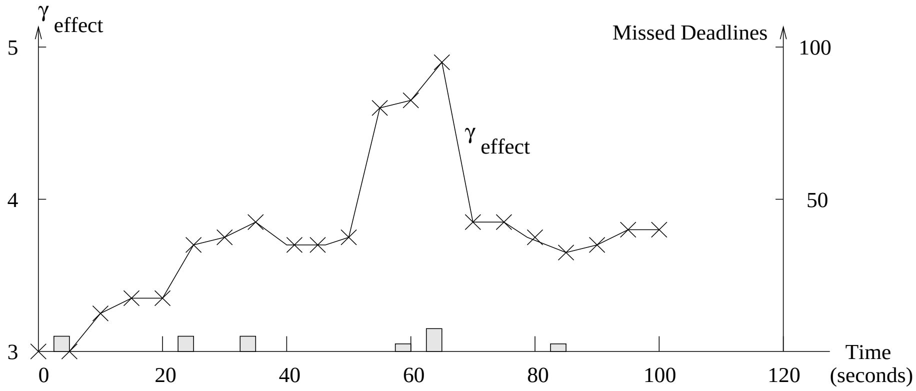

Figure 9: Improved responsiveness due to QoS level adjustment heuristic

set of acceptable QoS levels, rewards, and QoS violation penalty. We presented a thread-per-ANA architecture that enforces the QoS contract. We described an implementation of ANAs as $q$threads scheduled by the communication subsystem to maximize aggregate reward. Scheduling ANAs involves appropriate selection of a QoS level for each ANA in accordance with current resource load and capacity. We described a mechanism for QoS-level selection based on a combination of periodic optimization and fast heuristic adjustment. A reward optimization algorithm was presented that uses a dynamically computed estimate of per-packet handling cost to determine the QoS-levels. A simple heuristic was described that dynamically adjusts the selected QoS levels in response to transient load disturbances. We discussed appropriate mechanisms for overload and underutilization detection, as well as policies for adjusting QoS levels in response to such conditions. The impact of such mechanisms and policies has been briefly illustrated using experimental results from an actual implementation of the communication subsystem. The analysis shows that the design is capable of meeting its stated goals.

References

[1] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[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*, vol. 4, no. 1, pp. 1–10, February 1996.

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

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

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

[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] 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*, pp. 90–99, May 1994.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

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

[17] K. Nahrstedt and J. Smith, "The QoS broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, 1995.

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

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

[20] 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.

[21] 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.

[22] 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.