Adaptive Quality-of-Service Session Management for Multimedia Servers

John Reumann and Kang G. Shin

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

Abstract

Contemporary multimedia applications usually require servers to provide static QoS (Quality-of-Service), such as constant bit rate or fixed average transfer rate. These requirements do not reflect the application needs, but result from lack of support for adaptive QoS in applications as well as in operating systems. Multimedia servers should provide "acceptable" service under a variety of QoS configurations. When they are ported to systems that do not support explicit QoS guarantees (e.g., the Internet), the exploitation of multimedia's inherent adaptability is an absolute must.

We propose a model for server design that addresses the need for adaptive QoS. Using economic theory, the server is designed to allocate resources so as to maximize the weighted sum of clients-perceived QoS. Our model views the QoS defined for a session as a variable that can be changed on-the-fly to improve user-perceived QoS on an aggregated basis. This on-the-fly change capability allows the server to degrade or upgrade the QoS of individual sessions within a pre-specified range, depending on the loading condition and/or resource availability. Simulation of our model showed that it (i) increases the number of clients served by the system between 30 and 130%; (ii) improves clients' perception of the service; and (iii) effectively controls the number of adaptation operations needed for these improvements.

1 Introduction

A main disadvantage of today's multimedia server architectures is their inability to support load-sensitive yet "acceptable" Quality-of-Service (QoS). Conventional multimedia server architectures are based on making "binary" admission decisions: either admit a client while guaranteeing a specific level of QoS or reject him. This design may work well for small intra-nets with a few clients, e.g., in-house training videos. However, if one tries to serve a greater audience, one would need a solution that offers more flexibility in handling active sessions, i.e., degrades (upgrades) the QoS for each active client within a pre-specified range as the load on the server increases (decreases). Such flexibility will greatly benefit the design of on-demand services because it enables the server to provide degraded (upgraded) yet satisfactory service to many clients under heavy (light) load. As a result, clients may try to receive better service by avoiding peak load times. Finding an appropriate adaptation model that helps achieve this is very important. We propose such a model, combining adaptability with commercial applicability.

Multimedia servers should operate under various resource constraints, as they must usually adapt their clients' QoS to changes in resource availability. A client should be able to specify his preference of a particular QoS configuration over others. Under certain constraints these preferences can be translated into utility functions [4, 11]. Since utility functions provide a compact representation of preferences, we can use them in making resource-allocation decisions inside a multimedia server in a load-sensitive manner.

Evolution of integrated multimedia services will expand the domain of commercial multimedia applications. Server design will be influenced by both economic incentives (primary) and technical challenges (secondary). The primary objective is to implement the service provider's economical policies while providing service to the clients. Therfore, each multimedia server has to fit into its economical environment, rather than forcing the environment to adapt to yet another type of multimedia technology. More specifically, we want service policies defined in economic terms so as to translate them directly into concrete policies of client session management within the multimedia server, thus interfacing economic theory to the application domain by the means of price and utility. This will require some economic modeling.

Modeling every client of a multimedia server as its own independent economic entity would require too much of modeling overhead, especially if there are frequent client arrivals. We therefore present a model, which reduces economic theory to an acceptable tradeoff between the efficiency of representation and "natural feel" of the price–utility interface. The tradeoff chosen in our model is to consider groups of similar clients, instead of considering each client individually. Membership in such a group should be associated with a price/fee, so that clients have to evaluate whether the description of a group fits their interests and affordability; they may or may not subscribe to it, depending on their evaluation results. In this way, the decision problem of assigning a client to an "interest group" and determining his utility function is off-loaded to the client himself.

The problem of defining groups and the membership fee is entirely an economical decision, subject to technological limitations and market behavior. The impact of this decision extends far beyond the scope of multimedia systems. Hence we will assume that the service provider has come up with group definitions, including the groups' utility functions for different levels of QoS, probably derived from opinion polls or estimation. The provider will make these descriptions visible to its clients, so clients are aware of the price, QoS, and utility of the group they may subscribe to. The supported levels of QoS are coded into the actual implementation of the multimedia service and may depend on the particulars of the server's operating system (OS).

In real economic systems, resources are allocated adaptively to changes in the environment, such as the arrival or departure of clients, which, in turn, alter the amount of each available resource. If more resources become available, some clients will be given better QoS, thus increasing average user-perceived QoS. Conversely, if resource supply is reduced due to new clients and resource failures, some clients' QoS may have to be degraded. This calls for the need for flexible QoS: a client may receive at one of several "acceptable" levels of QoS.[1] Flexible QoS enables the server to adapt to fluctuations in resource supply, load, and different client preferences, but we do not know how to change the resource allocation inside the server for each client session as the environment changes.

We propose a new model called the **Fl**exible **Mu**ltimedia **S**ession **S**cheduling (**FluSS**), which facilitates session management for a multimedia server with multiple service classes. The model includes definitions of server-side, client-side, and an algorithm designed to re-allocate resources as new clients arrive and old ones leave. The objective of this algorithm is to increase aggregated utility, i.e., the sum of all clients' perceived utilities.

The **FluSS** model leaves price determinaton up to the service provider. It implements the simplest economic configuration interface by allowing the service provider to group clients, estimate their assumed utilities, specify the fee they are supposed to pay, and configure the service levels supported by the server.

The rest of the paper is organized as follows. Section 2 gives a comparative perspective of our approach relative to other similar approaches. Section 3 models the allocation-optimization problem, and presents a solution algorithm. Section 4 adapts this model to the demands on high-end multimedia servers. Section 5 discusses how to integrate the proposed mechanism into off-the-shelf OSs, and performance issues related to the model. The paper concludes with Section 6.

2 Related Work

The multimedia architecture developed as part of the *Pegasus* Project at Universities of Twente and Cambridge [1, 10] addresses several questions related to the problem treated in this paper. In particular, their focus on adaptive QoS supports our adaptive QoS server architecture.

Mullender and Sijben [10] presented an architecture for dynamic QoS multimedia applications. They emphasize the need for adaptive QoS when considering the transmission of continuous media. One of the main motivations for their implementation of the *Nemesis* kernel [1] was to provide resource protection for competing multimedia applications against each other and support for QoS adaptation on-the-fly. The Nemesis kernel implements the desired functionality by offering an easy-to-use thread abstraction that allows external triggers for changes in the main function of a thread. It also implements the management of resource access rights on a per-process basis. Since most of the work is done in threads, the application can build its own thread scheduling and adaptation mechanisms on top of resource shares granted on a per-process level. Process-level guarantees are very firm. Thus, unlike most other OSs, Nemesis' virtual resources can be assumed to have a fixed capacity and are much more predictable than the virtual resources in today's standard OSs. The strong support for resource

[1] This differs from "best-effort" service in that clients will receive predictable QoS by specifying their acceptable QoS levels and passing admission tests.

shares and on-the-fly adaptation allows us to consider the allocation problem at a higher level of abstraction. In this sense, our work is complementary to the *Pegasus* Project, since our model deals with on-line adaptation, assuming that resource reservation can be implemented and the server code — that implements different algorithms — can be changed on a per-thread basis. In Section 5 we will discuss to what extent it is possible to achieve our goals on standard OS without requiring any non-trivial change to the OS itself.

The solution by Real Networks Inc. for streaming continuous media over the Internet is widely deployed in broadcasting live contents over the Internet [9]. Their audio and video servers use statistical information about the end-to-end connection between their RealAudio server and the client to accomplish a stable continuous media connection. The scheme depends mainly on buffering at the client side to mask the effects of varying network loads on the client-to-server connection. This is one building block in load-sensitive QoS provision. Nevertheless, buffering will only help mask temporary changes on the client-to-server connection. If a connection has become a permanently congested or stays congested for an extended period, it would be desirable to upgrade the QoS for other clients whose connection could carry additional information.

RealAudio's allocation policy is what we refer to as the *Null* model or best-effort. It simply checks whether the client's request can be granted without considering the possibility of degrading existing connections for a new client. This leads to low system utility under light load and only a small number of clients served under heavy load, as we shall see in Section 5. Furthermore, RealAudio primarily supports broadcasting one stream to many clients, but it does not provide any powerful means to serve many clients with CD-player-like semantics (start, stop, jump to, skip), where every client-to-server connection is unique. Adaptation to overload, like giving higher-paying clients better service or maximizing the number of clients served, is not implemented.

A similar architecture — only for video — is the Oracle video cartridge [8]. It avoids the unpredictability caused by a platform that does not support QoS guarantees by running one and only one application on the platform. Since the Oracle video server is designed to be ignorant of the contents it is shipping to the clients, one cannot expect the server to dynamically adjust the QoS in an application-sensitive way. In fact, once a connection is established, the server "pumps" the video at a constant rate according to the parameters that were agreed upon at the time of connection establishment. At connection-establishment time, the client and the server negotiate the versions of the multimedia contents that the client will receive, thus providing some basic adaptability. Unlike Nemesis, Orcale's architecture does not allow for on-the-fly adaptation. Buffering is its only means to adapt to changes in the environment. As stated earlier, this is useful for dealing with transient overload conditions and transient network failures, but fails to increase the availability of service or maximize the client-perceived utility. In contrast to the RealAudio architecture, Oracle's video server is designed to support playback semantics. For this reason, we also find extensive on-demand video retrieval and interface capabilities all built into the server.

RealAudio and similar on-demand services would be more attractive if they could exploit system support for dynamic, utility-based session management (that we propose here).

3 The System Model

The service model plays a key role in solving the allocation problem as well as in implementing the service. The model also affects both the user's perception of the service and the provider's ability to configure the server. Any adequate service model must:

- Guarantee clients to receive service at one of the several QoS levels they specified,
- Allow the clients $\rightarrow$ QoS levels mapping to be changeable without terminating any ongoing session,
- Represent the fact that upgrading the QoS of clients may lead to different rewards, depending on the client's perceived utility,
- Optimize allocations with respect to the total reward to be accrued while accounting for the associated overhead,
- Permit service providers to make tradeoffs among maximal total reward, service availability, and average connection-establishment overhead; and
- Allow clients to define their perceived utility at various QoS levels.

We have developed an abstract model that meets all of the above requirements. The model can be broken down into three main parts: the service definition (*service model*), service perception (*subscription model*), and the service provision (*interaction model*). The service model enables the provider to express his server's capabilities to the OS or a middleware layer, depending on the implementation of the **FluSS** model. The subscription model enables the provider to specify service classes available to clients, including QoS levels and the corresponding fees. The interaction model specifies how the server should implement session setup, tear-down, and adaptation with respect to client arrivals and departures. Resource failures may be integrated into this model as well.

3.1 Service Model

The server's ability to maintain a given QoS level for an individual session depends on the availability of necessary resources on the server. At this point the mapping of a multimedia service to specific technology becomes relevant because it determines the amount of resources consumed by each client at a certain QoS level. Although this mapping is not trivial (as discussed in Section 4), here we will assume its availability. Furthermore, we want to scale resource requests with respect to the total amount of a resource available. For example, if there are one million CPU cycles per second to be allocated in "1000-cycle units," then one unit represents 0.1 % of the CPU resource. This simple representation helps separate resource-allocation algorithms from the particulars of the underlying technology. By not committing to a particular implementation of resource shares, we define an OS-independent, implicit representation of resource allocation. Our particular percentage resource shares are tailored to bandwidth allocation. The algorithm presented in Section 3.4.1 works directly with any additive resource measure. Otherwise, it only needs to be slightly modified. In general, to abstract from particular implementation issues, we define a *service class* as the aggregation of code that needs to be executed to achieve a certain level of QoS along with a measured amount of resources that it requires to satisfy some high-level QoS constraint.

Only if all resources were local then a service class would represent the resources needed at a local node. However, since a distributed collection of workstations may cooperate to service a client, a resource-reservation "vector" is used to represent resource percentages at all nodes involved. It has to account for all resources at all involved nodes that need to be reserved for service provision at a specific QoS level.

A multimedia server is assumed capable of providing a certain service in several different ways, and clients are assumed capable of receiving service at any of the QoS levels specified, unless they are above a certain maximal level, beyond which the client's resources are exhausted. This is reasonable, because clients who want to interact with a specific server are usually willing to install the necessary software on their system that will manage QoS adaptation. Furthermore, we are not arguing for improved client-server inter-operability but for QoS adaptation. Support for different service provisions must therefore be designed to increase the server's flexibility in dealing with client arrivals and departures, and resource failures.

Definition 1 (Service Class) *For a server that supports b service classes, service class $i \in \{0, 1, \ldots, b-1\}$ contains information about which part of the server code to run, its parameters, and the corresponding resource consumption on the server to maintain a session for each client receiving service in that class. In case of k resources and b service classes, we define the function* **rescon**: $\{0, 1, \ldots, b-1\} \rightarrow [0, 1]^k$, *with* $rescon(i) = (x_1, x_2, \ldots, x_k)^t$, *where* $rescon(i)$ *represents the amount of all resources needed to deliver service to a client in service class i (t represents transposition).*

Note that the definition of a service class alone does not imply anything about the user-perceived QoS, which can be achieved in many different ways. For example, to transmit a video, one may use more CPU cycles to process (e.g., compress or smooth) the video signal, thus reducing network-bandwidth requirements; or use less CPU cycles, thus requiring more bandwidth. The actual optimization problem turns out to be much easier when there are fewer such tradeoffs and the canonical preferences over *service classes* coincide with the order achieved by the $\leq$ ordering on the resource consumption vectors for each service class. An example will be given in Section 3.2.

3.2 Subscription Model

Having discussed client preferences over service classes, we now want to have a closer look at the client side of our model.

Clients can be classified into groups, each with similar preferences on the multimedia signal quality. By increasing the number of groups, we can make finer distinctions between user preferences. Thus, grouping clients together is not a serious limitation in dealing with multimedia clients. A group of similar preferences is called a *subscription class*. Such grouping is used in many real-world settings. For example, one can lease different-bandwidth lines from public network providers, each offered at a specific price/rate. This rate is the same for all clients who want to receive service at a particular level of QoS. In case of QoS upgrade or degrade, clients within a particular *subscription class* are

discriminated.

Most models that offer group distinction exert their admission control by identifying the utility a group has for different service classes and setting a charge rate for that group. The group membership charge rate regulates access to the service. On the client's end, his subscription represents agreement with the group's pre-defined preference relation.

Our admission-control scheme differs from dynamic pricing models because the price for a service is known in advance, but the actual QoS doesn't have to be. Such flexibility is not feasible in dynamic pricing models, because the possibility of receiving better service would already be figured into the price, so the client would be very dissatisfied if he were ever served at lower quality than what he paid for. This inflexibility is too limiting for adaptive services. Another argument against dynamic pricing as a means of admission control for multimedia services, is the prohibitive overhead of price negotiation. This is true, especially when the service is requested frequently, like audio on-demand.

For the above reasons, slowly-changing subscription prices (or membership fees) charged over many distinct service sessions appear to be the best choice. They allow more flexibility in changing the service class for each client during an active session. In general, prices and client preferences are external to the multimedia server. We will use a combination of prices and client preferences as a measure for the utility the provider receives from serving clients of a specific subscription class. The price of a subscription class corresponds to the utility a client has in order to be fully satisfied with the QoS he receives. When service to the client is degraded, his utility does not drop to zero immediately but decreases according to a relative utility function u with $Image(u) \subset [0,1]$ and the domain of all possible service classes. The relative utility helps the provider recognize how the client's utility reacts to changes in the service class.

Now, we can define the absolute utility function U (Fig. 1) as the product of the price of a subscription class and its relative utility function. We also ensure that the utility does not saturate after the client is being served at his "highest satisfaction" service class (u reaches 1), but we distribute an additional unit ("bonus") of utility equally over all remaining service classes, for which the client has sufficient resources. This is done to ensure that a client can receive better service than he pays for, if the server can provide it. There is one utility function u and one price for each subscription class.

Example: A service provider may decide to offer three subscription classes: CD quality, FM quality, FM or CD with natural preference for CD. Each of these will be priced differently. Assume CD quality costs \$50 per month, FM \$20 per month and the mixed subscription class \$40 per month. Assuming that the multimedia system can provide AM, record, FM, CD qualities — these are the service classes — we still need to define a relative utility function for each subscription class in addition to the price of a subscription class (Fig. 1(a)).

Our model relies heavily on the determination of clients' sensitivity to changes in such QoS as frame rate, resolution, etc., since we want to use the absolute utility function U (Fig. 1) defined for each service class, as part of an objective function for the evaluation of allocations, i.e., client $\rightarrow$ service class mappings.

Definition 2 (Subscription Class) *Suppose there are t active clients and the service provider offers v subscription classes, $s(i) \in \{0,\ldots,v-1\}$, and each client $i \in \{0,1,\ldots,t-1\}$ is associated with exactly one subscription class $s(i)$. Each subscription class j is specified by the price $price(j)$, its corresponding relative utility function u_j, and a timeout. The relative utility function is specified over all possible service classes offered by the multimedia server. "Timeout" specifies the amount of time for which a client of subscription class j is guaranteed to receive service. The price and timeout of a subscription class may be subject to change, whereas the utility function cannot be altered once a subscription class is instantiated.*

Note that these subscription classes are maintained on the server. Contracts between the provider and the client are established on the basis of particular instantiations of subscription classes. Subscription class instances should, therefore, closely model clients' real preferences. Furthermore, the server should devise the utility functions so that it may maximize aggregated system utility while providing high service availability.

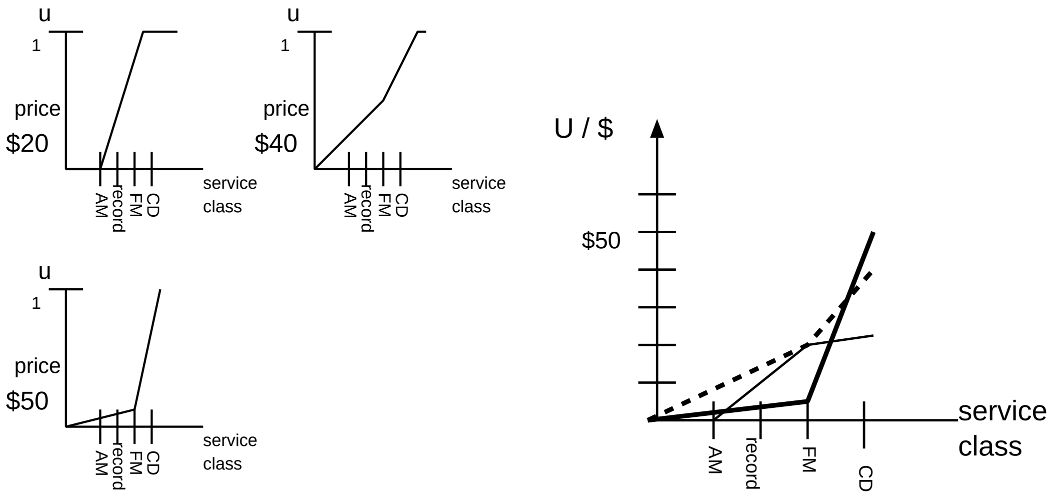
3.3 Subscription-Model $\rightarrow$ Service-Model Mapping

An *allocation* is a mapping of active *subscription class* members to service classes, and it must satisfy the following two conditions:

AC1: Each client receives service at exactly one QoS level specified by exactly one service class. No client receives service in the trivial class **null** (no service), unless the client's connection timed out. The function *sclass* that maps a client to a service class takes the client's ID as a parameter.

AC2: In case of t clients, the following constraint must be satisfied:

$$\sum_{i \in \{0,1,\ldots,t-1\}} rescon(sclass(i)) \leq (1,1,\ldots,1)^t$$



(a) Subscription class definition

(b) Derived absolute utility

Figure 1: Subscription class and absolute utility.

where $\leq$ is a vector comparison in which the left-hand side is less than or equal to the right-hand side in every component.

These conditions ensure that (i) an allocation specifies a service class for every client and (ii) resources will not be over-booked. Note that resource allocation is done implicitly by chosing a service class for a client. Resource constraints restrict the set of possible client→service-class mappings.

Our discussion of the model and the algorithm focused mainly on QoS adaptation to client arrivals and departures, but the model is general enough to accommodate other sources of change, such as the one caused by failures. In case of a resource failure, **AC2** may be violated and the allocation algorithm or the interaction model has to guarantee that **AC2** will be met as soon as possible, upon detection of the failure.

3.4 The Interaction Model

We now describe the core part of the model by formulating the problem to find an applicable solution. The resource-allocation problem turns out to be a slight modification of the well-known *knapsack optimization problem* [6]. Since the knapsack problem is known to be NP-complete, we will relax it somewhat for on-line use. If we strictly maximize aggregated system utility, the optimization problem has to be solved every time a client enters or leaves the system. (The same applies in case of resource failures.) The optimization problem for t clients is to *maximize* $\sum_{j \in 0,1,\ldots,t-1} u_{sj}(sclass(j))$ by changing the *sclass* mapping, subject to **AC1** and **AC2**.

If the dynamic programming is used, the time complexity is typically in $O(SIZE * M)$ where M is the number of items from which we can choose and $SIZE$ is the size of the knapsack. The items from the knapsack problem correspond to the service classes in our problem. Each client has to be served at exactly one of the service levels offered by the server. The profit function defined in the knapsack problem corresponds to the absolute utility of a service class. A more difficult problem is to identify the size of the knapsack. For simplicity, we will first look at the one-resource case in which it is easy to identify the size of the knapsack. (This will later be generalized to the case of multiple resources.) If the resource could be allocated at the granularity of $1/x$ ($x \geq 1$) relative to the total amount of a resource available, we can dedicate the resource to each client at x different consumption levels. The size of the knapsack would be x, which is usually not very large, but as we add one more resource, we can dedicate one more resource to each client. Assuming addition of one resource, we have to multiply the reciprocal of the granularity of the first resource by that of the newly-added resource to get the size of the knapsack for the extended problem. This implies an exponential growth of the problem in the number of resources considered. The exponential growth in the optimization space as the resource-allocation granularity becomes finer, makes the problem intractable. So, we propose an optimistic approach to solving this problem.

3.4.1 An Optimistic Approach to Solving the Allocation Problem

In the one-resource case, the problem is equivalent to the simple knapsack problem. Our algorithm acts just like the dynamic programming solution for this one-dimensional resource problem [5]. Unfortunately, its complexity is higher than in the classical case because we have to enforce the additional constraint **AC1**.

OptCycle:

1. `Pick the bottleneck resource for which the allocation will be optimized.`
2. `Pack the resource optimally for the first client under all levels of resource availability. Make sure that AC2 holds for every solution computed. Remember the maximum utility achieved for each allocation level of the resource considered.`
3. `Add next client to the allocation, and try to give him all possible shares of the considered bottleneck resource. Allocate the n-1 clients first by deciding the best utilization they can get out of the remaining resource share of the considered resource. Now, pick the best service class for the new class that is feasible. The optimal allocation is now computed by finding the maximum utility that can be achieved over all levels of resource allocation to the new client.`
4. `If there are more clients then go to Step 3 else output the optimal allocation.`

Step 1 of the *OptCycle* algorithm may be non-trivial but we assume that in most cases the heuristic of picking the most heavily-overloaded resources may work well. Note that there are sophisticated heuristics for finding the bottleneck resource but the main goal of the "optimistic approach" is to reduce the problem's inherent complexity.

3.4.2 Remarks on the Optimistic Approach

A simple example shows that the above simplification weakens its applicability: we do not allow network bandwidth to be traded for CPU bandwidth, and vice versa, because we have to consider the allocation of both resources to optimize utility. More specifically, the amount of network bandwidth allocated to one client may affect the maximum utility achievable for the current client, but our algorithm might only be looking at the CPU resource to increase utility. (The resource tradeoff problem will be addressed in a separate forthcoming paper.)

If it is possible to determine the critical bottleneck resource — overbooked and not a candidate for resource tradeoffs — then we are again facing the "simple" one-dimensional version of the knapsack problem because in this case **AC2** would kick in only for the bottleneck resource. Thus, packing the bottleneck resource optimally becomes the goal, which is exactly what the optimistic approach implements. Because real systems often encounter one bottleneck resource at a time, the optimistic approach will perform well in realistic settings.

3.4.3 Need for a Simpler Solution

Despite the significant complexity reduction achieved, one may question whether this algorithm will perform efficiently even if client arrivals are as frequent as 1 Hz and higher. To obtain an applicable solution, we need to adjust the model to fit the demands on multimedia servers, which should be able to support at least 100 clients per server. The admission algorithm should be done, on average, well within a second. *OptCycle* might not cope with such demands if resource granularity is too fine.

An important fact to keep in mind is that *OptCycle* exhibits pseudo-global characteristics. For example, it is possible that the service class of every client will be changed. This adjustment requires changing scheduling parameters and potentially switching to another service algorithm and client adaptation, inducing significant overhead. We will in the next section reduce this overhead and show how its reduction influences the quality of resource allocations.

4 Efficient Solution to the Allocation Problem

Since it is time-consuming for the *OptCycle* algorithm to reallocate client sessions, we introduce a *short-term repository* (STR), from which new clients draw some resources until their admission decision is made; the resources needed for the STR are reserved in advance. We propose use of a STR for two reasons. First, it is time-consuming to make an admission decision, as it may have to invoke an optimization process. Second, the number of clients served without a STR is less than that with it, because new client arrivals might not increase aggregated utility sufficient enough to be considered "admissible," but should not be rejected so as to keep the rejection rate low. We also introduce a "cache" for connections that have just been terminated, so that a new request may take the place of a client who has just left, without triggering a time-consuming optimization process.

4.1 Short-Term Repository and Free-List

Instead of waiting until the admission test completes, we will set aside a fixed amount of resources for newly-arriving clients. The STR represents resources that are reserved on the server but not used to serve existing clients. Its amount is so chosen to accommodate the mean number of clients arriving during the execution of the admission test. A newly-arrived client that cannot receive any more resources except for those allocated to him from the STR will reduce his requirements. If more resources become available, the STR will be replenished before making any other changes to the allocation. The downside of this order of steps is that the maximum aggregated utility for the server is reduced because the resources in the STR cannot be used to increase the utility of active clients.

Another way of admitting clients is to use a *free-list* (FL). Instead of degrading active clients' QoS to accommodate a new client, we may take advantage of the fact that a client of the same subscription class, thus, by definition, sharing the same preference relation, may have left the server a very short time ago. Thus, the new client can take the place of the client who just left without triggering any adaptation operation, thus reducing the number of times the *OptCycle* algorithm to be invoked, and decreasing the overall adaptation overhead. Every time the *OptCycle* algorithm is invoked, all resources on the FL will be made available for improving the service class of active client sessions.

4.2 Session Termination

Timeouts need to be implemented in the server; otherwise, a client could block others from accessing the server forever. In the OS context, this problem is typically referred to as starvation. Taking a timeout is the first step towards terminating an active client session. Upon expiration of a client's time quantum, he is put on a *droppable-list* (DL). Once on this list, the client's absolute utility will start to decline steadily to ensure that he will eventually be dropped to accommodate new clients. The introduction of a DL is reasonable since it would not make any sense to terminate a client session immediately if nobody can benefit from the freed resources.

The resources allocated to those clients, who do not resign themselves in time but leave or are dropped from the DL, are not put on the FL. If this were not the case, a client could resume immediately after being disconnected, as his resources would be "buffered" in the meantime. Of course, the most common source for session termination will be the client's "logging off." Once a session of subscription class i is closed, we will put the resources reserved for that particular session on the FL for subscription class i, where the reservation will be kept for a predefined time period or until a new arrival uses it.

4.3 Admitting New Clients

The most convenient way to admit a new client is to take resources from its subscription class' FL. So, checking the FL is the first step in the admission-control algorithm. If this step is successful, the algorithm takes and reserves the resources from the FL, allocates them to the new client, and exits.

Another straightforward way to allocate resources to a newly-arrived client is to satisfy its resource requirements directly from the DL. Clients on the DL will be dropped in increasing order of their maximum utility. All of the k lowest utility client sessions whose aggregated utility is less than the new client's maximum, will be dropped, as they contribute to the congestion of the server. More specifically, the ratio of reward received from maintaining those "expired" sessions to the resources these sessions consume is unacceptable.

If the admission algorithm fails to allocate the new client by using the above two steps, it allocates resources from the STR to it, which will suffice to start serving the client at the lowest service class defined for the server. We will then start degrading active client sessions for the new client. If there are no clients who, if degraded, lose less utility than the gain by upgrading the new client to the next higher service class and also free sufficient resources for the upgrade, we leave the new client at its current level. Otherwise, we upgrade its service class again.

4.4 Periodic Check

Since the proposed model uses timeouts "generously," we want to guarantee system integrity only after periodic system checks. If we were to deal with timers while serving the clients, it would be impossible to serve many clients simultaneously. Periodic integrity checks may lead to inconsistencies between checks, e.g., clients whose timeout has occurred could still be receiving full service.

Periodically we release all resources that were tied up on FLs for too long, drop clients on the DL whose utility is almost zero, and run *OptCycle* to upgrade clients' service classes. We also need to replenish the STR if the resources allocated for it do not match its specified size. The period of this check is an important design parameter, and is defined in terms of how many adaptation operations the greedy strategy may perform before *OptCycle* must be run again.

4.5 Analysis of the Hybrid Approach

Our choice of a mixture of greedy and global (*OptCycle*) strategies for the **FluSS** model is based on the comparison of algorithmic complexity of both optimization strategies. Despite its simplicity, *OptCycle* still has a strong negative influence on the system performance, which we were able to observe as the simulation speed declines. The time per simulation run greatly depended on the frequency at which *OptCycle* was invoked. This observation is backed up by our time complexity analysis.

The complexity of the *OptCycle* algorithm depends on two parameters: the number of clients, n, and the granularity of the server's resource, $g \in [0,\ 1]$. The number g represents the minimal amount of the resource that can be allocated to a client.

Since the dynamic programming is used to solve the allocation problem, a table of size $O(n/g)$ must be constructed. An entry, (n_i, g_i), represents the optimal allocation for all n_i clients who were given g_i of the bottleneck resource. When creating new entries, we have to ensure that **AC2** holds, possibly leading to a search through all $1/g$ rows of the previous column. However, this is unlikely, since we are assuming abundant supply of all resources, except for the bottleneck. Nevertheless, to guarantee the correctness of the algorithm, we still need to check **AC2**, which will have a significant impact on the worst-case analysis for the algorithm. Thus, for n clients the complexity of *OptCycle* is computed as the cost of building the table, which is $O(n/g^2)$.

For the greedy approach, in the worst case all n clients are at the lowest service level and can all be upgraded to the highest service class. Each upgrade operation step costs $O(\log(n))$ per-client, since we will keep clients on a priority heap, ordered by their potential utility increase and $O(\log(n))$ is the cost of maintaining the heap condition after each insertion. An insertion must be made every time a client's potential utility gain changes, i.e. after each upgrade operation. The greedy approach for degradation is defined analogously. In this way, we compute the complexity of $O(n \log(n))$. Note that the number of service classes only accounts for a constant factor in the complexity analysis, since we can upgrade each client by at most the number of service classes that the server offers.

5 Application and Evaluation of FluSS

This section describes the application scenario that inspired **FluSS**, presents the advantages of **FluSS** over the common practice in multimedia server design, and evaluates the effects of server design parameters.

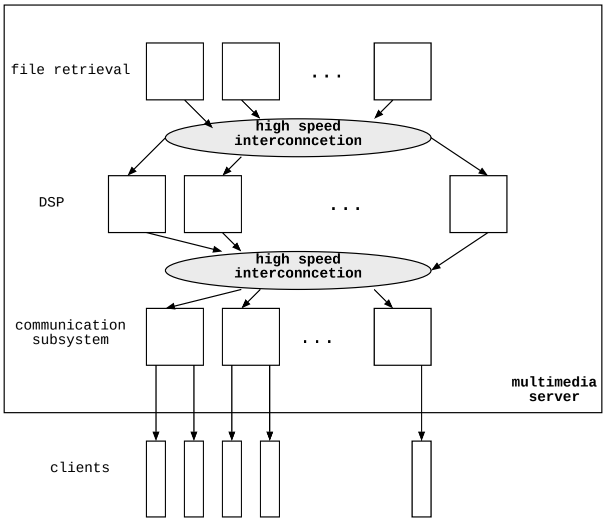


Figure 2: A multi-stage server.

5.1 A Multi-Stage Multimedia Server

In the *OptCycle* algorithm we made the implicit assumption that resources can be effectively allocated to individual server threads. This assumption doesn't hold for almost all of today's commercial OSs. The main difficulty lies with their inability of reserving resources, thus causing interferences among the server threads. For example, reservation of hard disk bandwidth for one thread will affect the amount of memory bandwidth available to another thread. The amount of cross-resource interference tends to be so significant that no simple solution for resource reservation may be derived and implemented. However, there is an exception: the problem of reserving/allocating CPU cycles is very well understood, solved and implemented in most modern OSs. POSIX [3, 12] defines the *de facto* standard for scheduling threads and enables the programmer to change the time-sharing policy for his threads from user-space. We propose to enhance the scheduling capabilities of POSIX-compliant OSs slightly to provide resource-reservation mechanisms necessary for **FluSS**, i.e., fair-share scheduling [7].

Our solution differs from conventional multimedia sever implementations in that, instead of integrating all functionalities into one node (workstation or PC), the service is broken up into functional components that would interfere with each other if they were run on the same node. So, we decided to distribute them over several nodes. Furthermore, each unit (Fig. 2) is built around one particular bottleneck resource. This design makes it easier to ensure that all resources, except the bottleneck resource, are abundant at the local node. The composite server functionality is achieved by combining the components in a pipelined fashion. The performance of this "pipeline" will depend on the performance of the slowest component in the chain. Finding and optimizing the slowest component in the chain guarantees the *Opt-Cycle* algorithm's optimality. Note that separate boxes in Fig. 2 may be either separate workstations or separate processing elements integrated on one board. The only requirement is that the interconnection network should not be the bottleneck for the server, i.e., should connect the individual stages of the server with sufficient bandwidth, and the processing units should not interfere with one another.

Decomposition is the first step in providing general resource-reservation mechanisms. The second step is to exploit the available resource-reservation mechanisms — ideally fair-share reservation — to reserve different shares of the bottleneck resources at the involved nodes. If fair-share reservation is unavailable for a particular resource we must resort to mapping fair-share resource allocation to fair-share CPU allocation. The idea behind this mapping is that the frequency at which a thread can consume resources depends on the CPU share it receives. Hence there must be a CPU quantum to be allocated to a thread so that it receives its reserved share of the bottleneck resource. Note that this step can be skipped if the node runs an advanced multimedia OS like Nemesis [1].

One may ask: "Given a desired thread performance, as defined by the **FluSS** model, how does one optimally configure the scheduling parameters for this thread in order to achieve the target performance?" Although the answer to this question is important, we will only briefly describe a potential solution (Fig. 3), since the derivation of a complete answer is beyond the scope of this paper.

Since OSs must handle many unpredictable events like interrupts, failures, branch-prediction in the processor, etc., one cannot determine the mapping analytically. Instead, one has to measure the performance experienced by the resource-consuming thread subject to different scheduling parameters, such as minimum time-slice, priority, etc. The thread should therefore be instrumented for efficient monitoring and feedback mechanisms, which will help find the optimal scheduling parameters for a desired level of QoS. Franken's thesis [2] presents a model that achieves the needed mapping by monitoring and stochastic modeling. If the scheduling is as simple as fair-share scheduling, which only depends on the parameters thread latency and CPU utilization, the mapping can be accomplished efficiently.

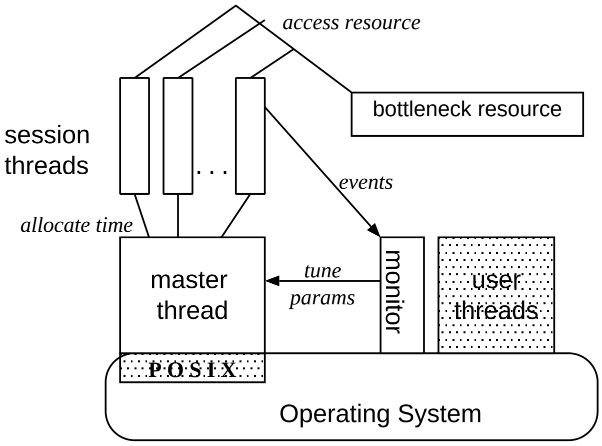


Figure 3: Potential node setup.

5.2 Simulating FluSS

We now want to evaluate the goodness of **FluSS** via simulation. The simulation results were very stable as the server activities were simulated for 27 hours, achieving less than 2% variations in our measurements of long-term averages. Due to their highly system-dependent nature, we determined the trend of utility, adaptation operations, and clients-in-service for different choices of system design parameters.

5.2.1 Simulation Parameters

To give a rough estimate of the performance of the proposed algorithm in realistic settings, we modeled the server using the information on RealSystem 5.0 [9], Real Network's audio-on-demand server. This widely-used system supports four service classes: 14.4, 28.8, ISDN, and Dual ISDN. According to Real Networks, it is possible to support more than 1,000 sessions per server PC at 14.4 kbps when broadcasting live contents. With individual streams the performance is significantly lower. Due to the unavailability of any more accurate information about the number of streams supported for "canned" contents, we modeled a server that can support up to 128 sessions at 14.4 kbps. This requires a bandwidth of 230 KB/s, which is manageable by a single workstation server. Table 1 shows the modeling parameters. We generated only one bottleneck resource, since the bottleneck-determination problem was not within the scope of this paper.

Furthermore, we defined the time granularity for the simulation to be 2 ms, so the server cannot handle client arrivals closer together than this. We chose 2 ms because it is a lower bound on the time in which we can receive a request for service, do the first "greedy" admission test,

serv. class ID	res. cons. vector
1	(0.0078125, 0.00390625, 0.00390625)
2	(0.015625, 0.0078125, 0.0078125)
3	(0.0234375, 0.015625, 0.015625)
4	(0.03125, 0.01953125, 0.01953125)

Table 1: Service class description

ID	$	Timeout/min	utility vector
1	15	45	(0.95, 0.98, 0.99, 1.0)
2	30	45	(0.3, 0.6, 0.98, 1.0)
3	50	45	(0.1, 0.6, 0.8, 1.0)

Table 2: Subscription class (SubCs) description. (utilities as utilities for each service class)

Load	SubCs1/s	SubCs2/s	SubCs3/s
light	0.005	0.02	0.01
low	0.02	0.07	0.03
high	0.035	0.1225	0.0525
heavy (overload)	0.05	0.2	0.1

Table 3: Arrival rates for different SubCs at different load levels.

Load	FluSS	Null
low	0.00	0.12
high	0.00	0.27

Table 4: Average rejection rates: **FluSS** versus **Null**

then generate a reply. This bound will be exceeded as soon as clients' workstations do not reside in the same workstation cluster as the server.

To get a handle on client inter-arrival times and session durations, we assume they are exponentially-distributed. This assumption is consistent with most queueing models of client arrivals. The parameter for session durations was the same for all service classes in all simulations and used the parameter 6×10^{-7} (a mean of 45 min). We evaluated the performance of **FluSS** under high, low, and very light loads. The corresponding parameters are given in Table 3.

The server resources considered in the simulation are CPU, hard disk, and communication (in that order of appearance in the resource vector). The CPU was considered the bottleneck resource because it is involved in signal enhancement, data retrieval, and protocol processing. Note that the bottleneck resource depends on the server setup.

The lowest quality is near AM radio with a 14.4 kbps connection. At 28.8 kbps one receives quality between AM and FM radio. Finally, ISDN facilitates true FM quality. Using both channels of ISDN will guarantee near-CD quality. Each of these QoS levels is represented by a service class. Clients are assumed to be capable to receive service at all of these levels.

The modeled server offers three different subscription classes. The lowest quality might be of interest to those who want to access voice transmissions, such as radio learning at 14.4 kbps for $15 per month. The second subscription class targets at those who want to listen to music at FM quality, ideally receiving service at 64 kbps. The cost is $30 per month. The last and most expensive class is for CD quality, and served best at 128 kbps. It costs a monthly fee of $50. See Table 2 for the exact utilities used in our simulation. The generated conditions for the simulation were mainly modeled after the motivating example in Section 3.2.

5.2.2 FluSS Enhances Service Availability

The **FluSS** model is compared against today's common practice in session management for multimedia servers that admits clients as long as there are sufficient resources to set up sessions for them. The server does not distinguish between clients, and simply tries to serve each client as best as it can. Furthermore, there is no concept of adaptation. Because of the model's simplicity we will refer to it as the **Null** model.

Intuitively, **FluSS** will outperform the **Null** model in terms of availability because it has the option of degrading clients' QoS, which is not supported by the **Null** model. What is surprising is the extent to which **FluSS** outperforms the **Null** model. Table 4 shows that the simple policy does not scale well to high-load environments whereas **FluSS** does. Lack of adaptability to load variations is "handled" by excessive over-design in current multimedia systems. By contrast, flexible QoS allows the service providers to tailor their systems more to their real needs, thus drastically decreasing the cost of the server and eventually the clients' cost.

Moreover, **FluSS** is found to be able to serve 30% more clients under low load than the **Null** model (40 as opposed to 30). Under high load this gap widens even more to 233% (70 versus 30). As we approach the overload region, the two policies drift apart even farther. The number of clients served in the **FluSS** model is sensitive to the removal of the short-term repository. Its removal decreases the average number of clients served by 10% and results in a non-zero rejection rate. This negative impact is amplified as load increases. Clearly, adaptation and the ability to admit clients on a "conditional" basis improve the server availability notably.

5.2.3 Allocation Improvements

Since we aim at improving the aggregated utility of allocations, we need to evaluate the utility improvements of **FluSS** over **Null** as well as the impacts of design parameters in **FluSS**.

As expected, **FluSS** outperforms the **Null** model (Fig. 4) by 15–25%, depending on the choice of system design parameters. There is a clear anti-proportional relation between the inter-*OptCycle* interval and the achieved aggregated utility. This is because the local strategy might get stuck at local extrema if the solution is not optimized from time to time by a global strategy. The decline in the aggregated utility is larger under higher load, to an extent that even the **Null** model achieves greater aggregated utility if *OptCycle* is not run frequently.

As expected, FLs have a negative impact on the achievable aggregated utility, as they were not introduced to increase utility but to decrease overhead. The aggregated utility with FLs ranks approximately 20% lower than without them (Fig. 5). this is significant with respect to the gain achieved so far, one must find a compelling reason for using FLs. The immediate decline of utility in the presence of FLs is due to the high arrival rates with which we have experimented. The lower the arrival rate the flatter will be the slope of the decline in utility in Fig. 5.

5.2.4 Trading Allocation Optimality for Less Overhead

Decreasing overhead effectively means improving the aggregated utility, since the resources saved on adaptation operations can be used to admit more clients to the server, which would then increase the aggregated utility. The overhead per session adaptation is not negligible, since it requires to change both the resource reservation and the the QoS level of a server thread.

Knowing that the aggregated utility decreases linearly with the time between *OptCycle* runs, we would like to confirm that the number of operations decreases faster so that we can make an effective tradeoff.

The decrease in the number of adaptation operations with the increase of inter-*OptCycle* intervals resembles a negative exponential function, meaning that its decline will be larger than a linear function at zero and less as we approach positive infinity. If we increase the time between *OptCycle* runs from zero to 15,000 adaptation operations, we notice a 10-fold difference in the number of adaptation operations (Fig. 6). The aggregated utility for the same interval only changes by less than 10%, indicating the need for trading optimality for a reduction of adaptation overhead.

We noticed in the discussion of aggregated utility that FLs have a negative effect (Fig. 5) on utility. This effect is compensated by the fact that while achievable aggregated utility decreases by 20%, the number of adaptation operations changes by more than 800% (Fig. 7), because every time a client is admitted through FL, adaptation is not necessary. Extending the time to keep resources on the FL does not decrease the number of operations any further beyond the optimal point, but it does not increase it either.

5.2.5 Finding the Optimal Choice of Parameters

FluSS requires a few parameters to be adjusted to enhance the performance of the multimedia server. Especially, the tradeoff between maximum utility and adaptation operations makes the choice of parameters difficult. So, we suggest to measure the number of CPU cycles consumed per adaptation operation. Once this number is determined it can be multiplied to the rate of adaptation operations and expressed in terms of resources consumed per second.

Now, to determine the utility gain from increasing the time between *OptCycle* runs or the introduction of free-lists, one has to figure out the resource consumption by adaptation operations executed at the new inter-*OptCycle* intervals. One can then compute the reduction in resource consumption achieved by the optimization step.

We compare the sub-optimality (in terms of utility loss) resulting from performing *OptCycle* less frequently against the average utility gain if the freed resources were used to upgrade clients' service classes. The expected utility gain can be determined by extending the simulation or a rough analytic approximation. By iterating this process the rough estimation may work well; otherwise, the parameters can be fine-tuned after the server is set up.

The choice of parameters is system-dependent, so it must be made for every newly-configured system. Our simulation must be fed with the parameters that fit the description of the new multimedia server and derive a proper set of parameters from the simulation results.

6 Concluding Remarks

The proposed **FluSS** model is found to have significant advantages over current best-effort schemes in terms of service availability and allocation. Flexible QoS is the

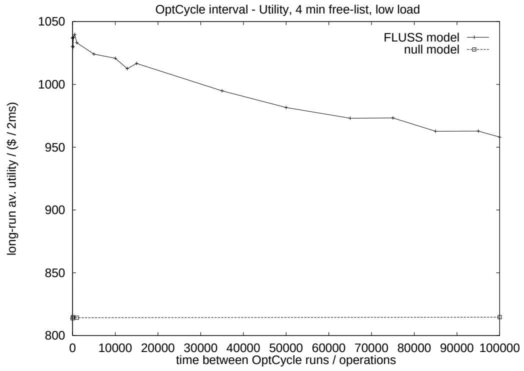


Figure 4. 4 min free-list, low load

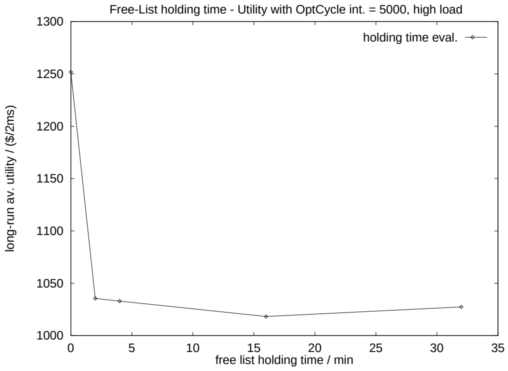


Figure 5. *OptCycle* interval 5000, high load

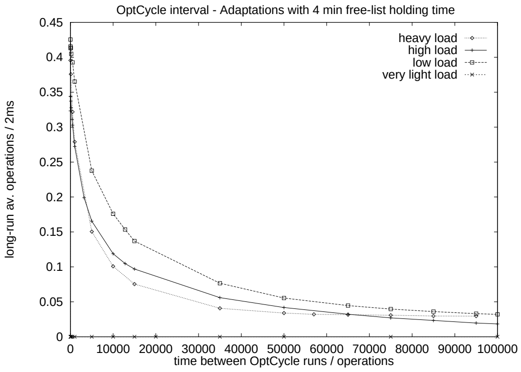


Figure 6. Four min free-list

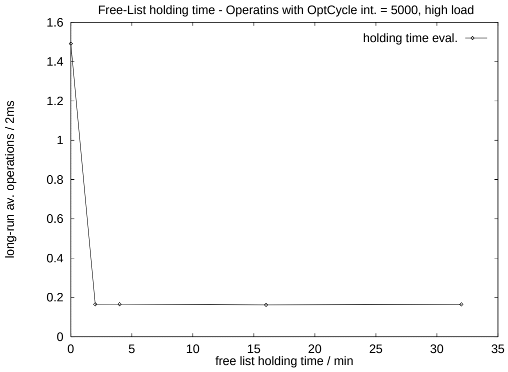


Figure 7. *OptCycle* interval = 5000, high load

key to building commercial multimedia servers. The resultant high service availability allows the provider to invest less money initially, because the server will still perform well even if it operates near its capacity. The provider can upgrade the server gradually to meet the need without excessive over-design. The fact that we were able to quantify the positive contribution of every component in the **FluSS** model is significant in that **FluSS**-like components can be used for adaptive session resource management, a key to future commercial multimedia systems.

We are currently exploring modifications of standard OSs to accommodate resource shares. This and the proposed session-management scheme together will eventually lead to building an advanced multi-stage multimedia server that supports adaptive QoS without requiring any non-trivial changes to commercial OSs. One of the design goals in building such a server is to use off-the-shelf OSs to show that, besides improving the service itself, adaptive QoS can also help reduce the cost of building high-end multimedia servers.

Acknowledgement

The authors would like to thank Hani Jamjoom of The University of Michigan for his contribution to the simulation tool and support during the early phase of this project.

References

[1] Robin Fairbairns, "Summary Report, Kernel Work Package", Esprit BRA 6865, University of Cambridge Computer Laboratory, 1993

[2] Leonard Franken, "Model-Based QoS Management", PhD. Thesis, Universiteit Twente, 1995

[3] Bill Gallmeister, "Posix.4: Programming for the real world", O'Reilly & Associates, 1995

[4] Reinhard John, "Definitionen und Sätze zur Vorlesung Entscheidungstheorie, Somersemester 1997", Rheinische Friedrich-Wilhelm-Universität, Bonn, 1997

[5] Thomas Leiserson, Charles Cormen, Ronald Rivest, "Introduction to Algorithms", New York, N.Y., McGraw-Hill, 1994

[6] Silvano Martello, Paolo Toth, "Knapsack Problems", Chichester, U.K., John Wiley & Sons Ltd., 1990

[7] Chen Lee, Raj Rajkumar and Cliff Mercer, "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach" , In *Proceedings of IEEE Multimedia Systems*, Japan, 1996

[8] Oracle, Inc., "Oracle Video Cartridge", Technical White Paper, www.oracle.com, 1997

[9] Real Networks, Inc., "Real Networks, the Home of Real Audio, Real Video, Real Flash", http://www.real.com/, 1997

[10] Paul Sijben, Sape Mullender, "An Architecture for Scheduling and QoS management in Multimedia Workstations", Pegasus paper 95-5, http://www.pegasus.esprit.ec.org/default.html, Universiteit Twente, 1995

[11] Hal Varian, "Microeconomic Analysis", 3rd Edition, W.W. Norton & Company, New York, 1992

[12] Fred Zlotnick, "The Posix-1 Standard: A Programmer's Guide", Redwood City, CA, Benjamin/Cummings Publishers, 1991