Retrieval Scheduling For Collaborative Multimedia Presentations*

Ping Bai† **B. Prabhakaran**‡ **Aravind Srinivasan**§

Abstract

The single-system approach is no longer sufficient to handle the load on popular Internet servers, especially for those offering extensive multimedia content. Such services have to be replicated to enhance their availability, performance, and reliability. In a highly replicated and available environment, server selection is an important issue. In this paper, we propose an Application-Layer Broker (ALB) for this purpose. ALB employs a content-based, client-centric approach to negotiate with the servers and to identify the *best* server for the requested objects. ALB aims to maximize client buffer utilization in order to efficiently handle dynamic user interactions such as skip, reverse presentation, go back in time. We also present details of a Collaborative Multimedia Presentation Platform that we have developed based on ALB.

Key words and Phrases. Scalable and highly available web servers, multimedia presentations, multimedia information retrieval schedule, buffer utilization and optimization.

1 Introduction

Scalability is a very important issue in providing Internet services, especially in view of the explosive growth in the number of Web users. When using the phrase *Internet services*, we consider text as well as multimedia presentations over the Internet, since accessing video and audio have become an integral part of the Web browsing activity. While this task of scalability can be considered as one of finding the best server or resource, the key issue of interest here is providing such a service in a transparent manner to the client. One such transparent approach is the Domain Name Server (DNS) proposed in [3, 9]. This approach uses a modified DNS to distribute incoming client requests

*A preliminary version of this work appears as a poster paper "Application-Layer Broker For Scalable Internet Services With Resource Reservation" in the *ACM Multimedia Conference*, November 1999.

†School of Computing, National University of Singapore, Singapore 119260. `baip@comp.nus.edu.sg`

‡School of Computing, National University of Singapore, Singapore 119260. Supported in part by National University of Singapore Academic Research Fund Grant RP 981669. `prabha@comp.nus.edu.sg`

§Bell Laboratories, Lucent Technologies, 600–700 Mountain Ave., Murray Hill, NJ 07974-0636, USA. Part of this work was done while at the School of Computing, National University of Singapore, Singapore 119260. `srin@research.bell-labs.com`

to different servers in a round-robin manner or based on a weighted classification. This allocation of client requests is done at the time of name-to-IP address translation carried out by the DNS. One limitation of this approach is that a translation request can go through a chain of intermediate DNSs that may cache the translation results. This caching might defeat the purpose of load balancing since an intermediate DNS can potentially allocate clients without the knowledge of the DNS in charge of load balancing. Another approach is to use a transparent front-end to forward requests to servers [2, 8, 3, 13]. Cisco's LocalDirector [2] is a product that uses address translation to forward requests to appropriate servers. The LocalDirector translates the headers of all data packets in both directions; however, the approach does not very well suit geographically distributed servers. IBM's NetDispatcher [8, 3] intercepts each IP packet that represents a new TCP connection. It forwards the connection and the subsequent data from a client to an appropriate server. Data from the server to the client, however, flows directly (i.e., not through the dispatcher).

An application layer *anycasting* has been suggested in [4], for locating services in the Internet. Each service provider has a unique Anycast Domain Name (ADN) that can be translated into a collection of IP addresses. The Anycast Name Resolver carries out the name-to-IP translation. This translation is based on the server membership for the particular requested service and the metric information associated with each member of the anycast group. The Anycast resolver collects and dynamically maintains performance metrics of each anycast domain member. An ADN resolver can have drawbacks similar to the DNS approach, due to the caching of ADN resolution results. The above-mentioned approaches are independent of the nature of the client request and how it might potentially load the server. For instance, a client's request might be for an hour-long multimedia presentation whereas another might be for accessing a text-based home page. These requests impose totally different loads on the servers. Also, a server selected based on the collected performance metrics may not be able to serve the request due to the nature of the request. For example, the selected server may not have sufficient disk bandwidth to admit a new multimedia presentation along with other requests being served. Content-based routing techniques have been proposed in [15, 17]. However, these techniques merely check whether a server is a owner of requested object(s), with the aim of improving the cache hit ratio.

1.1 Application-Layer Broker

One of the main features of future networks will be the providing of differentiated service to customers; i.e., guaranteed Quality of Service (QoS) will be offered. In particular, bandwidth reservation will be a possibility. Based on this scenario where networks offer QoS guarantees, we propose an *Application-Layer Broker* (ALB) architecture, as shown in Figure 1. ALB is at the application level (whereas other proposed architectures are more or less at the TCP/IP level where the decision-makers are not aware of the nature of client requests). ALB employs a content-based, client-centric approach. A client specifies the object(s) that is required and also when (i.e., the time) the object is required. ALB examines the contents of a client's request, negotiates with the possible sources of the requested objects, and identifies the times as well as the bandwidths at which the sources can deliver the requested objects. ALB then identifies the *best* source for the requested object(s) based on certain client-specific criteria, as discussed in Section 1.1.1.

Comparing the ALB proposal with those of others discussed earlier, ALB is at the application level with ALB being aware of the objects being requested by the clients. ALB also assumes advance

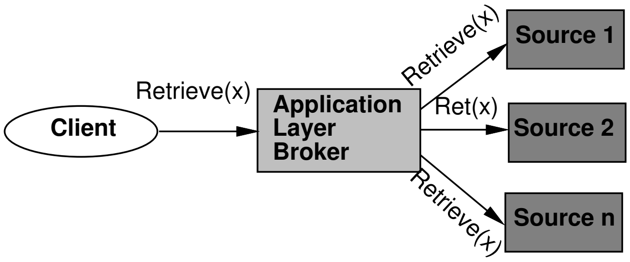

Figure 1: **Application-Layer Broker**

resource reservation features. Though these features may not be available widely in present networks, it is very clear that efforts such as Internet II are toward this direction. Also, decision-makers in other proposals *assign* requests to clients and hence the servers/sources do not have any choice. ALB negotiates with the sources by specifying the requested objects and hence the sources can admit the requests by determining the resources required (disk bandwidth, buffer etc.). Identification of the best source by ALB is carried out in a client-centric manner and obviously, this may not lead to balanced load across the involved sources. We, however, believe that this is natural since the aim of scalable architectures is to provide better response to clients and not mainly to balance the load across the servers involved. Since ALB negotiates with the sources by specifying the requested object(s), the sources can commit the required resources in a more precise manner. Hence, load imbalance among servers will not lead to performance degradation.

1.1.1 The ALB Approach

The ALB approach is oriented toward *long* multimedia presentations, since they impose more stringent load conditions on the servers, and since they are becoming common on the Internet. For instance, *www.broadcast.com* offers full movie presentations over the Internet. These long multimedia presentations are influenced by the amount of buffering done by the client system, since client buffering can help in reducing the number of network accesses for handling user interactions such as "reverse presentation" or "go back in time". With the increasing availability of memory and disk space, client buffering might look like a non-issue. However, multimedia presentations for long durations (say, an hour or more) can require storage space of the order of Gigabytes, depending on the resolution of objects and the compression techniques used. Client buffer can be critical for low-end client systems such as Personal Digital Assistants (PDAs) that can have memory space in the order of Megabytes. Also, the ALB approach is highly desirable for proxy-servers in WWW environments. Proxy-servers typically serve several clients' requests and hence disk space consumption can be huge. The approach followed by ALB is to help the client/proxy-server optimize its buffer usage during a multimedia presentation over the Internet. ALB chooses the sources for a multimedia object in such a way that the client/proxy-server maximizes the number of objects that can be held in its fixed-size buffer. Also, in the proposed algorithm, objects are *not* necessarily released from the buffer after their presentation time: they are allowed to stay in the buffer as long as possible. Furthermore, this algorithm effectively postpones the retrieval of an object as much as possible, i.e., it tends to retrieve each object as close to its presentation time as possible. These properties help in handling dynamic user operations such as *reverse presentation*, *go back in time*, and *skip time duration*. Furthermore, as mentioned above, efficient buffer utilization also helps us minimize network accesses when the user alters the presentation sequence.

ALB uses an algorithm that considers networks offering guaranteed Quality of Service. Briefly, for each server S_i and multimedia object O_j, we assume that the maximum instantaneous rate (i.e., throughput) at which server S_i can provide object O_j, has been negotiated. The main point is that we allow this rate to be an *arbitrary* non-negative function $r_{i,j}(t)$ of the time t. (Offered bandwidth typically varies over time, and hence it may not be realistic to assume that it is a constant. Also, $r_{i,j}(t)$ could be identically zero, indicating that S_i does not store O_j, or otherwise cannot offer O_j due to other service commitments.) Under such a general situation, we present an algorithm that provably postpones buffer overflow optimally. Also, the algorithm presented here can choose a source for an object (from a set of systems over which the object is replicated) in such a way that client buffer utilization is maximized. Please see Section 2 for the precise model; we just mention here that the problem is not trivial, since, for instance, the functions $r_{i,j}(t)$ are arbitrary. Suppose, for example, $r_{i,j}(t)$ is initially small and grows rapidly; how does it compare with the rate $r_{i',j}(t)$ (offered by another server $S_{i'}$) which is initially high but falls off rapidly? More importantly, we have to make decisions for *several* objects O_j simultaneously. If, for instance, we choose a server that provides an object very fast initially, one faces the issue of whether the other objects will have to be retrieved quite slowly in the beginning, in order to delay buffer overflow. However, as shown in Section 3, we can handle the situation, as well as some generalizations of it, well.

Organization of the paper: In the following section, we describe the system model and the basic problem. We present our retrieval algorithm and prove that it optimizes the utilization of available client buffer, in Section 3. In Section 4, we present strategies for the release of objects from the client buffer, in case of overflow. Handling dynamic user interactions during a multimedia presentation is described in Section 5. We have developed a Collaborative Multimedia Presentation Platform based on our algorithms, details of which are discussed in Section 6. In the final Section 8, we conclude our discussions.

2 System Model and the Basic Problem

A multimedia object in a document has to be available completely in the buffer before it can be presented. In the case of video objects, a video frame needs to be completely available before its presentation. (Subsequent frames can be retrieved in the inter-frame presentation time, if sufficient network bandwidth is available.) The system model and problem we consider here are as follows; we start with an informal description.

We have a collection of objects $O_1, O_2, \ldots, O_N$, each available at at least one (and possibly several) servers. We have to schedule a multimedia presentation, whose requirements are basically as follows. The objects have been partitioned into sets $A_1, A_2, \ldots, A_s$. For some given sequence of times $t_1 < t_2 < \cdots < t_s$ in the future, the objects in A_1 should be available in the client buffer at time t_1, the objects in A_2 should be available in the client buffer at time t_2, etc. (In other words, the client wishes to view the objects in A_i by time t_i, for each i.) Through negotiations with the servers, we know a certain maximum bandwidth at which each server S_i can provide each object O_j that it stores. We wish to come up with a retrieval schedule – one choice of server for each object, and retrieval rate that does not exceed the available bandwidth – in order to postpone buffer overflows as much as possible. Since the basic problem is the same for the retrieval of objects in each A_i, we now focus on the problem of retrieving objects in A_1 alone. We will be able to define this problem in an easier manner, and the algorithm we present can then be applied one-by-one for $A_1, A_2, \ldots, A_s$.

The formal problem of retrieving the objects in $A_1 = \{O_1, O_2, \ldots, O_n\}$ in order to optimally postpone buffer overflow, is as follows. (Further generalizations of this problem will be considered in Sections 3.2, 3.3, and 3.4.) We have a buffer of capacity B_{max}, measured in bits. The buffer currently contains some data, and hence is currently filled with capacity some B; so the available capacity is $B_{max} - B$. We have n objects $O_1, O_2, \ldots, O_n$, and the size of O_j in bits is b_j. All these n objects must be available in the buffer by some given time t_1 in the future; so we necessarily assume that $\sum_{j=1}^{n} b_j \leq B_{max}$ (otherwise, we cannot load all the objects into the buffer). We remark that if available buffer space or the offered network bandwidth is insufficient, we can consider reducing the *quality* of an object. The quality of an object can be modified by tuning parameters such as resolution. Modifying object qualities has been described in [1, 18]. We do not go into the details of object quality modification in this paper.

Now, furthermore, if we also have $\sum_{j=1}^{n} b_j \leq (B_{max} - B)$, then we should be able to load all these objects into the buffer without any buffer overflow, in which case retrieval can be done smoothly. But, it may happen that $\sum_{j=1}^{n} b_j > B_{max} - B$; in this case, we aim to postpone the inevitable buffer overflow as much as possible. We proceed to describe our problem further.

We have m servers $S_1, S_2, \ldots, S_m$, and we have a choice of retrieving each object O_j from some of the servers. Our objective is to judiciously choose one such server for each object O_j: our choice should be such that *buffer overflow* happens at the latest possible time before t_1. (The same server can be used to provide many of the objects if necessary.)

The above is the objective; what data on offered bandwidth do we have? Let the current time be t_0; recall that the time by which all the O_j must be available in the buffer is some given $t_1 > t_0$. We consider a general situation (which is particularly applicable for large retrievals) wherein the network expects to be slow during some time intervals, etc. For each server S_i and object O_j, we are given some arbitrary non-negative function $r_{i,j}(t)$, which gives the *maximum instantaneous rate* at which server S_i can provide object O_j for us. In other words, for any time interval $[a, b]$, the maximum number of bits of O_j that S_i can provide during this interval, is given by

$$\int_{t=a}^{t=b} r_{i,j}(t)\ dt. \tag{1}$$

In particular, if the function $r_{i,j}$ is identically zero, this means that S_i cannot provide O_j for us– maybe because it does not have O_j stored in it. We assume that each $r_{i,j}$ is continuous at all but some finite number of time instants t. (The discontinuities of $r_{i,j}$ model time instants at which the offered bandwidth changes abruptly, due to the termination or addition of traffic.)

(**Remark.** We work with the functions $r_{i,j}$ for now, as they represent the natural notions of maximum throughput and bandwidth. As shown in Section 3.4, certain scenarios do not immediately adapt to the model of the existence of functions $r_{i,j}$ for which the value in (1) is the maximum amount of

O_j that S_i can provide during the interval $[a, b]$. However, Section 3.4 shows that our algorithms directly carry over to several such more complicated scenarios also.)

While our algorithm below can handle any non-negative functions $r_{i,j}$, we point out that the two most realistic cases are:

(P1) Each $r_{i,j}$ is zero up to a point, after which it becomes some positive constant. That is, $r_{i,j} = 0$ for $t \leq T_{i,j}$; $r_{i,j} = a_{i,j} > 0$ for $t > T_{i,j}$. This corresponds to the server S_i being ready to offer O_j at up to a certain fixed bandwidth $a_{i,j}$ after a certain time.

(P2) This is a generalization of (P1), when each $r_{i,j}$ is a *step function*; this reflects the fact that maximum offered bandwidth is generally constant over short intervals of time. More precisely, for some given values $T_0 < T_1 < \cdots < T_p$, $r_{i,j}$ here is of the form

$$\text{for } k = 0, 1, \ldots, p-1,\ r_{i,j}(t) = a'_{i,j,k}, \text{ for all } t \in [T_k, T_{k+1}). \tag{2}$$

Here, $a'_{i,j,k}$ is some given numerical value associated with the function $r_{i,j}$. We will also assume that $T_0 \leq t_0$ and $T_1 \geq t_1$ for each $r_{i,j}$, i.e., that the function $r_{i,j}$ is well-defined for the entire interval $[t_0, t_1)$. (The values p, T_k and $a'_{i,j,k}$ could of course vary for different values of (i, j).)

3 Retrieval Algorithm

We now show how to optimally postpone buffer overflow, given arbitrary non-negative functions $r_{i,j}$. Consider any i, j. Now, if

$$\int_{t=t_0}^{t=t_1} r_{i,j}(t)\ dt < b_j,$$

then even if we try to retrieve O_j from S_i at the maximum rate $r_{i,j}$, we will be unable to retrieve all of O_j by time t_1; in such a case, we say that S_i is *irrelevant* for O_j. If this is not the case, then we say S_i is relevant for O_j. Clearly, in trying to choose a server for O_j, it is sufficient to consider only those S_i that are relevant for O_j.

For each j and for each S_i that is relevant for O_j, we first compute a value $s_{i,j} \in [t_0, t_1]$ that satisfies

$$\int_{t=s_{i,j}}^{t=t_1} r_{i,j}(t)\ dt = b_j. \tag{3}$$

Thus, $s_{i,j}$ is the time at which we should start retrieval, if we decide to *retrieve O_j from S_i at peak rate $r_{i,j}$, so that O_j is just available at the required time t_1*. Even for arbitrary non-negative functions $r_{i,j}$, an approximate value for $s_{i,j}$ can be computed efficiently by bisection search as follows. Since $r_{i,j}(t)$ is non-negative, the function

$$g(y) \doteq \int_{y}^{t_1} r_{i,j}(t) dt$$

is non-increasing as a function of y. Hence, $s_{i,j}$ can be estimated quickly by binary search in conjunction with a simple numerical integration procedure; in practice, it should be even simpler, since the most common families of functions are those given by (P1) and (P2). (The integration procedure for these is straightforward.)

The following lemma shows the crucial role played by the $s_{i,j}$. Suppose we have committed to some server $S_{f(j)}$ for each O_j. Given this, we just require a scheduling algorithm: for each j, at what (time-varying) rate at which to retrieve each O_j from $S_{f(j)}$, in such a way that: (i) the rate never exceeds the bandwidth bound $r_{f(j),j}$, (ii) O_j is available by time t_j, and (iii) buffer overflow, if any, is postponed as much as possible. Given any "server commitment" function f, a natural scheduling algorithm is the *greedy* one, which starts retrieving each O_j at time $s_{f(j),j}$ and then continues to retrieve at peak rate $r_{f(j),j}(t)$. The following lemma shows that for any given server commitment function f, we might as well employ this greedy algorithm, if we wish to postpone buffer overflow.

Lemma 3.1 *Suppose we have committed to some server $S_{f(j)}$ for each O_j. Given* any *scheduling algorithm $\mathcal{A}$, let the amount of buffer use at time t be $buf_{\mathcal{A}}(t)$. Let the buffer use at time t for our greedy algorithm be $B(t)$. Then, for all $t \in [t_0, t_1]$, $B(t) \leq buf_{\mathcal{A}}(t)$.*

Proof. Suppose that in algorithm $\mathcal{A}$, some object O_j is *not* scheduled by the above greedy approach. We will now show that if we change the retrieval of object O_j to the greedy manner, then the buffer requirement at any time will be no more than $\mathrm{buf}_{\mathcal{A}}(t)$. Repeating this for each such object O_j, we will end up with the greedy schedule, thus proving the claim.

Suppose O_j is not retrieved by the greedy approach; its retrieval must have started at some time $x_j \leq s_{f(j),j}$. For each t that lies in the interval $[x_j, t_1]$, let $g(t)$ denote the number of bits of object O_j that have been retrieved by time t by $\mathcal{A}$. Note that $g(t_1) = b_j$. Let $h_j(t)$ denote the instantaneous rate at which O_j is retrieved by $\mathcal{A}$, at time t.

Now, for all $t \in [x_j, s_{f(j),j}]$, object O_j would not have been retrieved at all until time t by the greedy schedule; so if we change O_j's retrieval to the greedy schedule, the buffer requirements will be no more in comparison with $\mathcal{A}$, up to time $s_{f(j),j}$. Next, for any $t \in [s_{f(j),j}, t_1]$, we have

$$b_j = g(t_1) = g(t) + \int_{u=t}^{u=t_1} h_j(u)du \leq g(t) + \int_{u=t}^{u=t_1} r_{f(j),j}(u)du; \tag{4}$$

hence, $g(t) \geq b_j - \int_{u=t}^{u=t_1} r_{f(j),j}(u)du$. But, for the greedy schedule, the number of bits of object O_j that have been retrieved by time t is **exactly** $b_j - \int_{u=t}^{u=t_1} r_{f(j),j}(u)du$. Thus, whatever scheduling strategy $h_j(t)$ we used for O_j, the amount of O_j retrieved at any time by $\mathcal{A}$ is at least as high as that retrieved by the greedy algorithm. By doing this one-by-one for each O_j, we conclude the proof. ∎

Figure 2 describes an example of the bandwidth-greedy algorithm, and also illustrates a different algorithm that retrieves at less than the maximum rate. The shaded regions represent the number of bits (of the object O_1) accumulated over a period of time. As can be seen from the figure, retrieval at less than the maximum available bandwidth accumulates more number of bits at any point in time than retrieval using the bandwidth-greedy algorithm.

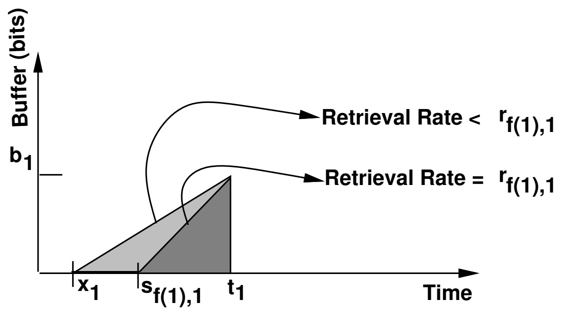

Figure 2: **Buffer-Optimality of Greedy Algorithm: Example**

3.1 The function Retrieve$(\cdot)$ and the algorithm

A key idea of our algorithm is to introduce an additional parameter x. We wish to define a function Retrieve(x), which takes as input a time $x \in (t_0, t_1]$. If there exists a choice of one server for each object (i.e., a way of retrieving all the objects by time t_1) in such a way that there is no overflow until time x, this function returns "Yes", and also outputs such a choice of one server for each object; if there exists no such choice, this function returns "No". Clearly, given such a function, we will simply need

$$x_0 \doteq \sup\{x \in (t_0, t_1] : \text{ Retrieve}(x) \text{ returns "Yes"}\}. \tag{5}$$

How can the computation of Retrieve(x) be done? Lemma 3.1 shows that whatever server we decide on for each O_j, we may as well use the greedy scheduling algorithm. Thus, suppose we choose some server S_i for O_j (clearly, S_i must be relevant for O_j). Then, since we may assume that we will employ greedy scheduling, the amount $\text{Ret}_{i,j}(x)$ of O_j retrieved by time x is given by:

$$\text{if } s_{i,j} \geq x, \text{ then } \text{Ret}_{i,j}(x) = 0; \quad \text{if } s_{i,j} < x, \text{ then } \text{Ret}_{i,j}(x) = \int_{t=s_{i,j}}^{t=x} r_{i,j}(t)\ dt.$$

The simple Observation 3.1 will be useful. This observation states that to check if overflow occurred at or before time x, it suffices to just check if the buffer is overfull *at* time x. This is true because the available buffer capacity remains constant (i.e., $B_{max} - B$).

Observation 3.1 *For any given assignment of servers to objects, there is no overflow* until *time x if the buffer is not more than full* at *time x.*

If our aim is to choose one relevant server for each object and then use greedy scheduling in order to not have the buffer more than full at time x, what choice of servers should we make? The answer is easy: for each O_j, simply choose a relevant server S_i which minimizes $\text{Ret}_{i,j}(x)$. It is easy to see that if at all we can avoid overflow until time x, this choice of servers will help avoid such overflow. Note the interesting point that once we fix x, we can in fact *independently* come up with a "good" choice of server for each O_j.

Thus, RETRIEVE(x) is as follows. For all (i,j) such that S_i is relevant for O_j, we compute $\mathrm{Ret}_{i,j}(x)$. (This can be done with sufficient accuracy via numerical integration.) For each O_j, define: (i) $\mathrm{opt}(j,x)$ to be the minimum, computed over all S_i that are relevant for O_j, of the value $\mathrm{Ret}_{i,j}(x)$; and (ii) $\mathrm{best}(j,x)$ to be any server S_i that is relevant for O_j and for which $\mathrm{Ret}_{i,j}(x) = \mathrm{opt}(j,x)$.

Our above discussion now shows that RETRIEVE(x) just needs to do the following. If

$$\sum_{j=1}^{n} \mathrm{opt}(j,x) \leq (B_{max} - B), \tag{6}$$

then RETRIEVE(x) outputs "Yes" and returns the server $\mathrm{best}(j,x)$ for each object O_j; otherwise if (6) is false, RETRIEVE(x) outputs "No". This completes the description of the function RETRIEVE(x).

Finally, recall that our main algorithm simply needs to find x_0 as in (5), and return the output of RETRIEVE(x_0). We can discretize this problem by allowing an absolute error of at most ϵ in the value of x_0; ϵ is a given error parameter here, which can be specified by the user. (An approximation such as this is reasonable since in reality, the functions $r_{i,j}$ will themselves be approximations of the actual offered bandwidths.) Define $M = \lceil (t_1 - t_0)/\epsilon \rceil$. We divide the interval $[t_0, t_1]$ into M equal intervals, each of length at most ϵ, by defining

$$t'_0 = t_0;\ t'_1 = t_0 + (t_1 - t_0)/M; \cdots\ t'_i = t_0 + i \cdot (t_1 - t_0)/M;\ \ \cdots;\ t'_M = t_1.$$

Thus, our algorithm needs to find the largest value k such that RETRIEVE(t'_k) returns "Yes", and to output the corresponding choice of servers. While this can be done by running RETRIEVE(t'_ℓ) for all $\ell = 1, 2, \ldots, M$, there is a faster method. Suppose k is the largest value of ℓ such that RETRIEVE(t'_ℓ) returns "Yes". Then, it can be checked that for all $\ell = 1, 2, \ldots, k-1$, RETRIEVE(t'_ℓ) will return "Yes"; also, for all $\ell = k+1, k+2, \ldots, M$, RETRIEVE($t'_\ell$) will return "No". So, k can in fact be found within $\mathrm{O}(\log M)$ invocations of RETRIEVE($\cdot$), via binary search.

Theorem 3.1 *For any given available buffer capacity $B_{max} - B$, any collection of non-negative bandwidth functions $r_{i,j}(t)$ and any parameter ϵ, we can efficiently find a retrieval schedule in which buffer overflow (if any) takes place within ϵ time of the latest possible overflow time. The running time of the algorithm depends multiplicatively on* $\log(1/\epsilon)$.

3.2 Extensions to dynamically changing buffer-size

In Section 3.1, we assumed that the available buffer capacity in the interval $[t_0, t_1]$ was a constant at $B_{max} - B$. However, it could be that the available buffer space varies dynamically due to the addition and deletion of other objects. We model this situation by assuming that the available buffer capacity, as a function of time, is given by a function $B_{avail}(t)$. (Theorem 3.2 summarizes our results in this direction.)

There is an interesting case to which Theorem 3.1 generalizes to. Suppose the buffer is intermittently fed with other data also; i.e., suppose that $B_{avail}(t)$ is an arbitrary *non-increasing* function.

(In particular, $B_{avail}(t)$ could be a constant, as in Section 3.1.) Observation 3.1 holds even for this case, since buffer space is only shrinking. So, it can be seen that Theorem 3.1 holds even under *any* non-increasing function $B_{avail}(t)$: this is part (i) of Theorem 3.2.

What about the case of arbitrary functions $B_{avail}(t)$ that can grow and shrink over time? We cannot employ Observation 3.1: just because the buffer was not overfull at time x, there is no guarantee that there was no overflow until time x. Indeed, an overflow could have happened at time $x' < x$; $B_{avail}(t)$ may then have increased sufficiently in the interval $[x', x]$ so that the buffer was not overfull at time x. The case of general functions $B_{avail}(t)$ and $r_{i,j}(t)$ looks difficult to handle in full generality. Nevertheless, even if $B_{avail}(t)$ is an arbitrary given function, we can solve the problem to optimality if the functions $r_{i,j}(t)$ come from a family of functions that generalizes the function family (P1) introduced in Section 2; we now show this.

To define this family, we start with some definitions. Recall the definition of $s_{i,j}$ from (3), and the corresponding notion of a server S_i being "relevant" for O_j. Suppose S_i and $S_{i'}$ are relevant for O_j, with $i \neq i'$. We say that i *dominates* i' *wrt* j iff: (a) $s_{i',j} \leq s_{i,j}$, and (ii) for all $t \in [s_{i,j}, t_1]$, $r_{i',j}(t) \leq r_{i,j}(t)$. We call the functions $r_{i,j}(t)$ *well-behaved* iff for each j, there is some i^* such that: (a) S_{i^*} is relevant for O_j, and (b) for all $i \neq i^*$ such that S_i is relevant for O_j, i^* dominates i wrt j.

We show in part (ii) of Theorem 3.2 that even if $B_{avail}(t)$ is arbitrary, we can solve the problem as long as the $r_{i,j}$ are well-behaved. Before that, we first show that the family of well-behaved functions is rich in the sense that it includes the practically important family of functions (P1) that we defined in Section 2.

Lemma 3.2 *Suppose all the functions $r_{i,j}$ come from the family (P1) of Section 2. Then, the $r_{i,j}$ are well-behaved.*

Proof. Recall the notation of (P1). Suppose S_i is relevant for O_j. Then, it is easy to check that $s_{i,j} = t_1 - b_j/a_{i,j}$; also, $s_{i,j} \geq t_0$ and $s_{i,j} \geq T_{i,j}$, since S_i is relevant for O_j. Define i^* to be any index such that S_{i^*} is relevant for O_j, and such that $s_{i^*,j} = \min\{s_{i,j} : S_i \text{ is relevant for } O_j\}$. Then, it is simple to check that for all $i \neq i^*$ such that S_i is relevant for O_j, i^* dominates i wrt j. ∎

Theorem 3.2 *(i) Theorem 3.1 generalizes to any non-increasing buffer-availability function $B_{avail}(t)$ also. (ii) Suppose $B_{avail}(t)$ is arbitrary. If the bandwidth functions $\{r_{i,j}\}$ are well-behaved, then for any parameter ϵ, we can efficiently find a retrieval schedule in which buffer overflow (if any) takes place within ϵ time of the latest possible overflow time. The algorithm's running time depends multiplicatively on* $\log(1/\epsilon)$.

Proof. Part (i) was proved above. For part (ii), consider any O_j. By the definition of "well-behaved"ness, there exists some i^* such that S_{i^*} is relevant for O_j, and such that for all S_i that are relevant for O_j, i^* dominates i wrt j. We denote such an i^* by $f^*(j)$.

Suppose there exists some algorithm $\mathcal{A}$ for retrieving all the objects by time t_1, such that for some given time $x \in [t_0, t_1]$, there is no buffer overflow until time x. We now show that such a retrieval is possible if we choose server $S_{f^*(j)}$ for each O_j, and then employ the greedy schedule introduced before. The proof is similar to that of Lemma 3.1, and is as follows. Suppose O_j is not retrieved by the greedy approach; let the notation $\mathrm{buf}_{\mathcal{A}}(t)$, x_j, $g(t)$ and $h_j(t)$ be as in the statement and proof of Lemma 3.1. Suppose O_j is retrieved from some server S_i by $\mathcal{A}$; we have $x_j \leq s_{i,j} \leq s_{f^*(j),j}$. (The last inequality follows from the definitions of domination and of $f^*(j)$.) Thus, as in the proof of Lemma 3.1, we need only show that if we retrieve O_j in the greedy manner from server $S_{f^*(j)}$, then for any $t \in [s_{f^*(j),j}, t_1]$, the amount of O_j retrieved is at most $g(t)$. Analogously to (4), we have

$$b_j = g(t_1) = g(t) + \int_{u=t}^{u=t_1} h_j(u)du \leq g(t) + \int_{u=t}^{u=t_1} r_{i,j}(u)du \leq g(t) + \int_{u=t}^{u=t_1} r_{f^*(j),j}(u)du;$$

the last inequality follows since $f^*(j)$ dominates i wrt j. So we have $g(t) \geq b_j - \int_{u=t}^{u=t_1} r_{f^*(j),j}(u)du$. But, for the greedy schedule, the number of bits of O_j that have been retrieved by time t is exactly $b_j - \int_{u=t}^{u=t_1} r_{f^*(j),j}(u)du$. Thus, we may as well retrieve each O_j from $S_{f^*(j)}$, and employ the greedy schedule. Given this fact, function RETRIEVE(x) is straightforward to implement, and hence the main retrieval algorithm can be designed through $\mathrm{O}(\log M)$ invocations of RETRIEVE($\cdot$), just as in the proof of Theorem 3.1. ■

3.3 Fuzzy Presentation Times

Recall that our assumption all along has been fixed object presentation times, i.e., that the user wishes to see the set of objects A_i at time t_i, for $i = 1, 2, \ldots, s$. However, object presentation times can be flexible: the user may tolerate slight variations in viewing times, as long as all such variations lie within guaranteed bounds [1]. In particular, suppose we can actually display the set of objects A_i at any time in the interval $I = [t_i - \delta_i, t_i + \delta_i]$. We can take advantage of this flexibility in further postponing buffer overflow, or in optimizing some other such objective function. Since our algorithm essentially computes an optimal schedule for each possible viewing time lying in I, our framework above can be easily extended to such situations. Briefly, we first discretize the interval I into sub-intervals of length ϵ' for a suitably small ϵ'. For each of the $1 + 1/\epsilon'$ discretized points t'_i in I, we compute the latest overflow time as described above; we may then choose an "optimal" point t'_i by which to load all of A_i into the buffer, using any other desirability criteria that we may have.

3.4 Lesser assumptions about the system

For $a < b$, let $\mathcal{N}_{i,j}(a, b)$ denote the maximum amount of O_j that S_i can provide during the interval $[a, b]$. As remarked in Section 2, there are settings in which there may not exist any functions $r_{i,j}$ such that for all a, b, $\mathcal{N}_{i,j}(a, b)$ equals the value (1). For instance, suppose a server S_i requires a *seek time* $\alpha_{i,j}(a)$ before initiating a request for O_j that arrives at time a. (This seek time may depend on the value of a, since it could be influenced by the prior commitments of S_i at time a.) Then, we have

$$\mathcal{N}_{i,j}(a, b) = 0 \text{ if } a + \alpha_{i,j}(a) \geq b; \text{ otherwise, } \mathcal{N}_{i,j}(a, b) = \int_{t=a+\alpha_{i,j}(a)}^{t=b} r_{i,j}(t)\ dt. \tag{7}$$

It can be verified that under such circumstances, there may exist no collection of non-negative functions $R_{i,j}$, such that $\mathcal{N}_{i,j}(a,b)$ equals $\int_{t=a}^{t=b} R_{i,j}(t)\ dt$ for all a, b. Hence, assuming the existence of functions such as $r_{i,j}$ may be questionable in some situations. However, this is not a problem for our algorithms: note that the only type of value they require is $\mathcal{N}_{i,j}(a,b)$, which they compute using (1). Thus, all we require is that the functions $\mathcal{N}_{i,j}$ are known (e.g., through a formula such as (7)).

4 Buffer Release

When buffer overflow occurs, some of the object(s) already resident in the buffer need to be evicted. We can release objects based on some known techniques such as *least recently used* (LRU) objects or more sophisticated caching algorithms using randomization [5]. We now propose another such algorithm for evicting objects from the buffer, in situations where we wish to keep the total size of evicted objects as small as possible. This is useful in situations where we do not expect much correlation between different object requests, and where buffer capacity is a valuable resource.

To make our problem more precise, note that the buffer needs to have all the currently requested objects $O_1, O_2, \ldots, O_n$, which have total capacity $B_0 = \sum_{j=1}^{n} b_j \leq B_{max}$. We wish to retain "as many" of the remaining set S of objects in the buffer, without violating the buffer's capacity: i.e., the total capacity given to S is $B_1 \doteq B_{max} - B_0$. There are two natural objective functions here, to formalize the above-seen phrase of "as many": (i) choose a subset S_1 of S with as large total capacity as possible, so that the total capacity of S_1 is at most B_1 (so we can pack these objects into the buffer); or (ii) remove a subset of smallest total capacity from S, so that the remaining objects have total capacity at most B_1. From the viewpoint of optimization, these two problems are equivalent: $S_1 \subseteq S$ is an optimal solution for (i) iff $S_2 = (S - S_1)$ is an optimal solution for (ii). However, note that (i) is just the NP-hard *knapsack* problem [6]; hence, both (i) and (ii) are NP-hard. Thus, one could look for approximation algorithms: *efficient* algorithms that provide solutions to within *guaranteed* factors of optimal. For (i), we can employ the provably-good approximation algorithms of, e.g., Lawler [10]. However, note that maximization and minimization problems that are equivalent in terms of optimization, often display substantial differences in terms of approximation [7]; we provide a provably-good approximation algorithm for problem (ii) now.

Our approximation algorithm for (ii) is simple and easily implementable. We keep the objects of S in sorted (nondecreasing) order of their capacities. Suppose the objects of S in this order are $O'_1, O'_2, \ldots, O'_m$, with the respective capacities of $b_1 \leq b'_2 \leq \cdots \leq b'_m$.

Case 1: $b'_m > B_1$**.** In this case, we remove all objects O'_i and stop.

Case 2: $b'_m \leq B_1$**.** Let i be the *smallest* index such that $b'_{i+1} + b'_{i+2} + \cdots + b'_m \leq B_1$ (such an index i exists; since $b'_m \leq B_1$, $i \leq m-1$). There are two subcases here.

Case 2a: $(b'_1 + b'_2 + \cdots b'_m) - b'_i \leq B_1$**.** In this case, we only remove object O'_i.

Case 2b: $(b'_1 + b'_2 + \cdots b'_m) - b'_i > B_1$**.** In this case, we remove the objects $O'_1, O'_2, \ldots, O'_i$.

This completes the description of the algorithm. It is easy to check that after the object removals as described above, the remaining objects have total capacity at most B_1.

We now analyze this algorithm and show that the total capacity of objects removed is at most twice the optimal amount OPT. We start with an obvious lower bound on OPT:

$$OPT \geq (\sum_{i=1}^{m} b'_i) - B_1. \tag{8}$$

Suppose Case 1 holds, i.e., $b'_m > B_1$. Thus, since O'_m cannot be stored in the buffer, we have $OPT \geq b'_m$. Adding this bound with (8), we get

$$2 \cdot OPT \geq (\sum_{i=1}^{m} b'_i) + b'_m - B_1 > \sum_{i=1}^{m} b'_i,$$

the last inequality following from the assumption that $b'_m > B_1$. Thus, the total capacity of removed objects in this case, $\sum_{i=1}^{m} b'_i$, is at most twice the optimal amount OPT.

Next, suppose Case 2 holds. If $i = 0$, there is nothing to prove, since we will not evict any objects. So suppose $i \geq 1$. Let $C = b'_1 + b'_2 + \cdots + b'_{i-1}$. By definition of i, we see that $b'_i + b'_{i+2} + \cdots + b'_m > B_1$; so, by (8),

$$OPT \geq (\sum_{j=1}^{m} b'_j) - B_1 = C + (\sum_{j=i}^{m} b'_j) - B_1 > C. \tag{9}$$

Now suppose Case 2a holds. Since $b'_i + b'_{i+1} + \cdots + b'_m > B_1$, at least one of the objects $O'_i, O'_{i+1}, \ldots, O'_m$ must be removed. However, each of these objects has capacity at least b'_i, due to our sorting. Thus, removing just object O'_i is an optimal solution in this case.

Finally, suppose Case 2b holds. Then, even if we remove object O'_i, we cannot accommodate all the remaining objects in the buffer; thus, $OPT > b'_i$. Adding this with (9), we see that the evicted total capacity $\sum_{j=1}^{i} b'_j = C + b'_i$ is at most $2 \cdot OPT$.

Thus, we have shown that our simple greedy algorithm always removes at most twice the minimum capacity that needs to be removed. In practice, we can make an enhancement to the basic algorithm above; this has been implemented in the Collaborative Multimedia Presentation Platform that we describe in Section 6. In the case where $b'_m > B_1$, if we have $b'_1 + b'_2 + \cdots + b_{m-1} \leq B_1$, then we need only remove object O'_m. This additional step further improves the quality of our buffer-release algorithm.

5 Handling Dynamic User Interaction

Presentation of retrieved multimedia information can be modified by operations such as reverse presentation, skip, and scaling the speed. These operations modify the sequence of objects to be presented and perhaps their times and durations of presentations. Hence, the network bandwidth requirements for the multimedia presentation may get modified. A new set of multimedia objects might have to be retrieved from their sources, causing additional network accesses to be made. In this section, we will discuss how our algorithm handles these operations by minimizing the number of network accesses that need to be made.

Reverse Presentation: This operation directs the sequence of a multimedia presentation backward in the time-line. Objects that were already played need to be presented again, according to temporal relationships in the reverse direction. The proposed client buffer-optimal algorithm helps in maximizing the number of objects that can be held in the client buffer of given size. Hence, it minimizes the number of network accesses that need to be made for handling reverse presentation operation. For objects that are not available in the client's buffer, the RETRIEVE algorithm can be invoked to schedule their retrieval.

1. The first step is to stop the arriving objects (if any) as they will not be useful for reverse presentation.
2. Present objects in the client buffer according to deadlines in the reverse direction.
3. Invoke algorithm RETRIEVE for objects that are not in the client buffer.

Skip Time Interval: Here, the client has to start viewing objects that will be presented (in the normal presentation sequence) after a specified time interval SI. For this, the set of multimedia objects that need to be presented after the interval SI are identified. If this set of objects are part of playing objects (PO) or arriving objects (AO), then the client can start the presentation once the objects are available. For objects that are not available in PO and AO, the RETRIEVE algorithm can be invoked. It should be noted here that the RETRIEVE algorithm postpones retrieval of multimedia objects as much as possible. Hence, it helps in minimizing the number of objects that are retrieved but *not* presented because of the skip interval operation.

Scale Speed: Here, the duration of presentation objects have to be scaled up or down by the given factor F. The presentation times t_i of objects will also be modified based on F. The proposed RETRIEVE algorithm already uses the maximum available network bandwidth. However, since the times and durations of objects presentations are modified by the scale speed operation, the arrival time of objects need to be computed again.

6 A Collaborative Multimedia Presentation Platform

Function RETRIEVE described earlier helps in identifying the source (in a replicated environment) as well as the retrieval request time for an object composing a multimedia document in such a way that the client buffer use is optimal. This approach also helps us to handle user interactions such as reverse presentation and skip time interval. It is easy to see that the RETRIEVE function is generic in nature, dependent only on parameters such as object presentation time, the available network bandwidth to each of the object sources, and the available client buffer. Hence, we have developed a

Collaborative Multimedia Presentation Platform (CMPP) that incorporates the RETRIEVE function. CMPP provides services such as scheduling the retrieval of objects composing a multimedia document, as well as user interaction services such as skip, reverse presentation, and scale presentation speed. CMPP assumes that the network service provider offers guaranteed Quality of Service (QoS) and advance reservation features, as shown in Figure 3. CMPP adopts a two-phase approach for identifying the retrieval schedule. In the first phase, CMPP identifies and *locks* (i.e., temporarily reserves) the network bandwidth to each of the object sources. CMPP uses the RETRIEVE function to determine the best object source and retrieval time. In the second phase, CMPP either *commits* network bandwidth to the selected object source (for the determined time interval based on the identified retrieval schedule) or it releases the network bandwidth (that has been temporarily locked) for other object sources.

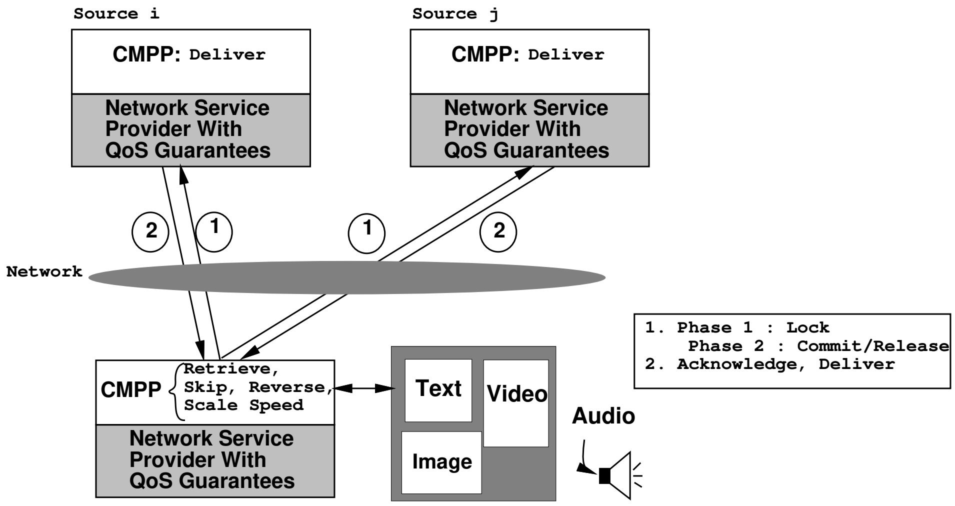

Figure 3: **Collaborative Multimedia Presentation Platform**

Implementation Experience: We implemented some of the techniques discussed above over a 100 Mbps Ethernet network of Sun Ultra Sparcs. The stored multimedia documents carry information about wild-life and the corresponding audio presents the sound made by each animal. Each multimedia page is composed of objects such as text, image, and audio. CMPP has been implemented as a Java application using Java JDK 1.2beta4, as this version supports the playing of audio in Java applications. Clients can be Sun workstations (running Solaris OS) or PCs running Windows 95 (or Windows 98), with support for Java JRE1.2 or JDK 1.2. CMPP, on the client side, is composed of classes such as *presentation, resolve, retrieve,* and *releasebuffer*. The *presentation* class includes use of windows for delivering multimedia objects and handling user interaction such as skip and reverse presentation. User interactions are interrupt-driven and hence the *presentation* class is essentially multi-threaded. The *resolve* class retrieves presentation specification: number of objects, their spatial and temporal characteristics. It also helps in interacting with an object's sources to find out whether an object can be delivered by a particular source or not. The *retrieve* class implements the RETRIEVE algorithm and identifies the best source and the retrieval time, with the objective of maximizing the client buffer utilization. The *releasebuffer* class implements the buffer release algorithm. Every time a new object is to be retrieved, this class checks the available space in the client buffer. If necessary, objects are released from the client buffer using the buffer release algorithm discussed earlier in the paper.

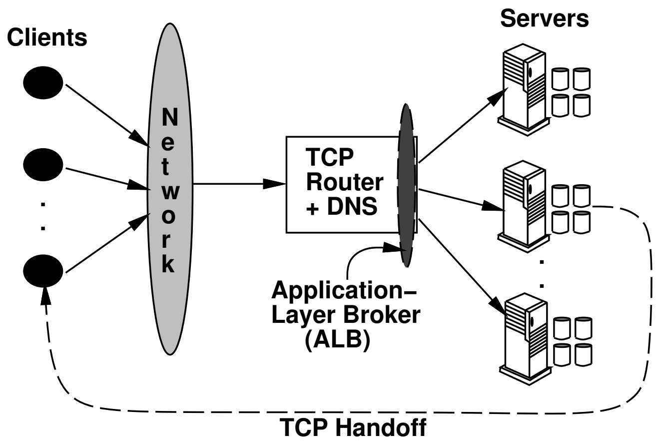

Figure 4: **ALB in Multiserver Web Environment**

The present implementation of CMPP is over TCP/IP. We are considering the possibility of implementing CMPP over networks that can offer guaranteed QoS in environments such as ATM. In the current implementation, the average execution time for the function RETRIEVE is 750ms. This time includes the two-phase process of identifying the available network bandwidth and then making the decision on the ideal source and retrieval start time for each object. In the current implementation, client buffer size was given an upper limit of 7 MBytes. With this upper limit, user interaction such as reverse presentation resulted in an average of 30% less network accesses. One can find more information about CMPP by visiting the URL *http://cram.iscs.nus.edu.sg:8080/~baip/cmpp*. ALB can be integrated into a multi-server web environment to select an appropriate server that can potentially respond to a client's request. For this purpose, we can employ a TCP router that uses ALB along with domain name services (DNS), as shown in Figure 4. The router should be capable of handing over the TCP connection to the chosen server in a manner that is transparent to the client. We are in the process of carrying out this idea.

7 Related Work on Retrieval Schedule Generation

ALB determines objects' retrieval schedules from possible servers and uses them to identify the *best* server from the point of view of optimizing client buffer utilization. In this section, we describe approaches presented in the literature on retrieval schedule generation for multimedia presentations. The derivation of retrieval schedules for distributed multimedia presentation has been studied in many works, such as [12, 11, 16, 18]. In [12], the presentation of multimedia objects is based on a Petri nets description of the temporal specification. The retrieval schedule is derived by assuming a certain bandwidth to be provided by the network service provider. Based on the derived retrieval schedule and the assumed network bandwidth, estimates for the buffer resource requirements on the client system are made. However, the proposed algorithm does not check whether the estimated buffer resources are available or not. Also, it does not handle replication of objects on multiple sources.

In [11], Li *et. al.* use time-flow graphs to capture interval-based *fuzzy* presentation schedules, and synchronization of independent sources. Their algorithms guarantee that there will be no gaps in the source's information delivery schedules. However, they do not address the issues of object replication on multiple servers, constraints on resources such as network bandwidth and client buffer. As in [12], in [16], the authors use Petri net model to describe temporal specifications, and they base the retrieval schedules for single object source on the fixed presentation schedules. In [18], Thimm *et. al.* describe a method which adapts the presentation schedule to the changes in the resource availability by modifying the overall quality of the presentation. They also do not consider object replication over multiple servers.

In [1], the authors have proposed a flexible retrieval schedule algorithm for single object sources. We can also deal with flexible presentation schedules as shown in Section 3.3, and deal with replicated objects on multiple servers. [14] addresses the problem of buffer sharing at the media source for maximizing the utilization of disks and buffer. In our approach, we maximize the number of played objects that can be held in the buffer at each stage, on the client side. This helps in handling user operations such as reverse presentation.

8 Summary and Conclusion

Popular Internet servers offering extensive multimedia content often have to be replicated for improving their availability and performance. A client request to such highly available and scalable servers have to be routed to an appropriate server. The methodology employed for server selection is an important criteria that will influence the performance of the system. In this paper, we have presented a content-based, client-centric approach, *Application-Layer Broker (ALB)*, for this purpose. ALB examines the contents of a clients request, negotiates with the possible servers for the requested objects, and identifies the *best* server to deliver the object. Identification of the *best* server is done in such a way that the client can maximize its buffer utilization. ALB also helps in postponing objects' retrieval as close to their presentation time as possible. These features help clients to handle dynamic user interactions such as skip, navigate in time, and reverse presentation. We have proven that the greedy-approach used by ALB is optimal in terms of client buffer utilization. We have used ALB to develop a Collaborative Multimedia Presentation Platform (CMPP). We are in the process of integrating ALB in an Internet environment by incorporating features such as name services and connection-handoff protocol.

Acknowledgements. We thank Debasis Mitra, Yong-Chiang Tay, and the referees for their helpful suggestions.

References

[1] K. Selçuk Candan, B. Prabhakaran, V.S. Subrahmanian (1996), *CHIMP: A Framework for Supporting Distributed Multimedia Document Authoring and Presentation*, Fourth ACM International Multimedia Conference, Boston, Editors: T.D.C. Little and W. Hall, ACM Computer Society Press, NY, pp. 329-340, 1996.

[2] Cisco Local Director, *http://www.cisco.com/warp/public/751/lodir/in dex2.shtml*, CA, USA, 1998.

[3] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Web Server", COMPCON 1996, Santa Clara, CA, USA, IEEE Computer Soceity Press, Piscataway, NJ, USA, pp. 85-92, 1996.

[4] Z. Fei, S. Bhattacharjee, E.W. Zegura, and M.H. Ammar, "A Novel Server Selection Technique for Improving the Response Time of a Replicated Service", Proceedings of INFOCOM'98, San Francisco, CA, USA, IEEE Computer Soceity Press, Piscataway, NJ, USA, 1998.

[5] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator and N. Young, *Competitive paging algorithms*, Journal of Algorithms, Vol. 12, 1991, pp. 685-699.

[6] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman and Company, New York, 1978 (printing updated in 1991).

[7] D. S. Hochbaum (Editor), *Approximation Algorithms for NP-Hard Problems*, PWS Press, Boston, MA, 1997.

[8] IBM Corporation, IBM Interactive Network Dispatcher. http://www.software.ibm.com/network/dispatcher/, NY, USA, 1998.

[9] E.D. Katz, M. Butler, and R.A. McGrath, "A Scalable HTTP Server: The NCSA Prototype", Computer Networks and ISDN Systems, Vol. 27, pp. 155-164, 1994.

[10] E. L. Lawler, Fast approximation algorithms for knapsack problems, *Mathematics of Operations Research* **4**, 339-356, 1979.

[11] L. Li, A. Karmouch and N. D. Georganas (1994), *Multimedia Teleorchestra With Independent Sources: Part 1 and Part 2*, ACM/Springer-Verlag Journal of Multimedia Systems, Vol. 1, No. 4, 1994, pp. 143-165.

[12] T.D.C. Little and A. Ghafoor (1992), *Scheduling of Bandwidth-Constrained MultiMedia Traffic*, Computer Communication, Butterworth-Heinemann, July/August 1992, Vol. 15, No. 6, pp. 381-388.

[13] M.V. Mohamed-Salem, G.V. Bochman, and J. Wong, "A Scalable Architecture for QoS Provision in Electronic Commerce Application" Electronic Commerce Project, University of Waterloo, 1999.

[14] R. T. Ng and J. Yang (1996), *An Analysis of Buffer Sharing and Prefetching Techniques for Multimedia Systems*, ACM Multimedia Systems, Vol. 4, No. 2, 1996, pp. 55-69.

[15] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers", Proceedings of the Eighth Internation Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA, USA, Editors: A. Agarwal, ACM Computer Society Press, NY, USA, 1998, pp. 205–216.

[16] S.V. Raghavan, B. Prabhakaran and Satish K. Tripathi (1996), *Handling QoS Negotiations in Orchestrated Multimedia Presentations*, Journal of High Speed Networking, Vol. 5, No. 3, 1996, pp. 277-292.

[17] J. Song, E.L. Abegnoli, A. Iyengar, and D. Dias, "A Scalable and Highly Available Web Server Accelerator", Technical Report RC21377, IBM T.J. Watson Research Division, January 1999.

[18] H. Thimm and W. Klas (1996), *δ-Sets for Optimal Reactive Adaptive Playout Management in Distributed Multimedia Database Systems*, 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, IEEE Computer Soceity Press, Piscataway, NJ, USA, Feb. 1996, pp. 584-592.