

Odyssey: A High-Performance Clustered Video Server

Min-You Wu

Department of Electrical and Computer Engineering
The University of New Mexico
Albuquerque, NM 87131-1356
Email: wu@ece.unm.edu
Phone: (505)277-1078
Fax: (505)277-1439

Wei Shu

Department of Electrical and Computer Engineering
The University of New Mexico
shu@ece.unm.edu

Chow-Sing Lin

Department of Information Management
Southern Taiwan University of Technology, Taiwan
mikelin@mail.stut.edu.tw

Odyssey: A High-Performance Clustered Video Server

Min-You Wu, Wei Shu

Department of Electrical and Computer Engineering
The University of New Mexico

Chow-Sing Lin

Department of Information Management
Southern Taiwan University of Technology, Taiwan

Abstract— Video servers are essential in Video-on-Demand and other multimedia applications. In this paper, we present our high-performance clustered CBR video server, *Odyssey*. Odyssey is a server connecting PCs with switched Ethernet. It provides efficient support for normal play and interactive browsing functions such as fast-forward and fast-backward. We designed a set of algorithms for scheduling, synchronization, and admission control, which results in a high utilization of resources. Odyssey is able to deliver a large number of video streams.

keywords— video content retrieval, clustered video servers, real-time scheduling, Quality-of-Service, interactive browsing operations.

1 Introduction

Rapid advances in computer, storage and network technologies make it feasible and imperative to build video servers capable of providing a large variety of services, ranging from Video-on-Demand (VoD) for home entertainment, digital library for information retrieval, to distance learning for education. This on-demand service must satisfy users in terms of its high availability, various browsing functions, and quick responsiveness. Increasing demand on the capacity of video servers makes parallel video servers inevitable. Limited resources must be fully utilized to increase availability to users. A stream can switch between normal play and browsing functions at any time without interfering any other service streams. And the responses to these interactive operations must be fast enough to make users more productive and have an enjoyable experience.

Expensive high-end engines have been used to provide on-demand availability, while their browsing functionality and responsiveness have yet to be fully realized. Following the development footprints of supercomputers, parallel computers, and clustered computers, it is believed that using a cluster of low-cost machines in parallel to provide the large-scale on-demand service is an inevitable direction. We developed

a clustered parallel video-on-demand server architecture with deterministic service guarantee. This high-performance, low-cost clustered constant-bit-rate (CBR) video server, *Odyssey*, is a server connecting PCs with switched Fast Ethernet. It provides efficient support for normal play and interactive browsing functions such as *fast-forward* and *fast-backward*¹. A set of algorithms for scheduling, synchronization, and admission control results in an efficient utilization of resources.

Some important issues are identified in providing availability and functionality for retrieval of stored media. The number of deliverable streams should be maximized with the same amount of available resources. Instead of relying on statistical multiplexing, this design meets the challenge by precisely allocating, scheduling, and coordinating storage, network, and processor resources to maximize the resource utilization. Providing high availability is hard; at the same time, providing full browsing functions with fast response is even harder, since resources are fully utilized while many browsing functions raise different requirements of resources. We investigated how to preserve the high availability and at the same time provide a full spectrum of browsing functionalities without requiring additional resources. Three important topics for clustered multimedia servers are:

- scheduling of multiple resources;
- providing complete interactive browsing functions; and
- providing quick response to user's commands.

Although each of these can be researched separately, integrating all of them into a clustered server becomes a real challenge. We propose a unified model to solve these problems. By taking full advantage of the cluster capacity, we focus on developing the system-level infrastructure and techniques that enable future networked multimedia systems to be effectively and efficiently constructed, paving a way to achieve the goal of scalable high-performance VoD and digital library engines.

Several research projects investigated techniques for designing video servers. Related issues include disk layout and scheduling strategies, admission control policies, real-time support, etc. [1, 2, 3, 4, 5, 6]. Most research assumed a single-node system; other research projects investigated the design techniques for parallel video servers [7, 8, 9, 10, 11, 12, 13]. Video data striping schemes have been studied in [14, 1, 2, 12, 13]. However, many systems rely on statistical multiplexing, which cannot truly guarantee quality of service. System resources cannot be fully utilized without a good scheduling strategy. If the video streams are not arranged properly, a large buffer space is required, which introduces long delay [15]. A greedy stream scheduling algorithm for clustered architectures has been proposed [7]. Our approach aims at precise scheduling of video streams to maximize system throughput and to minimize delay and buffer usage. As an example, the *Tiger* project [12] uses the wide striping strategy to balance the load. However, it does not have a conflict-free stream scheduling algorithm to avoid unnecessary delay. After the system

¹Fast-backward is often called *rewind*.

load reaches 55%, the delay suddenly increases to more than 13 seconds [12], while in our system, when the system load reaches 90%, the delay is about one second and rejection rate is less than 3%.

Interactive play can be accommodated by displaying frames at a rate higher than the normal play [16]. Displaying frames at a higher speed increases bandwidth consumption. Bandwidth smoothing techniques are used to solve the bandwidth problem for interactive operations [17]. It provides the fast-forward and fast-backward capacity within a VCR-window by using large buffers. Scan outside of the VCR-window requires renegotiation. The bandwidth requirement can be reduced by skipping blocks, such as one proposed in [18]. Different data layout schemes for interactive playout have been described in [19]. Our prefetching algorithm and grouping algorithm provide formal models for browsing functions on clustered servers. A stream can switch between normal play and others without requiring additional system resources. Skipping frames will provide VCR-quality browsing, however, it is difficult to implement, especially on parallel servers.

There have been little research on minimizing the response time of browsing functions on parallel clustered servers. Reddy has proposed a method to reduce the response time by scheduling urgent requests with a higher priority [20], which can be applied to a sequence of short video clips. We have developed methodologies to minimize the delay for general interactive browsing functions.

We present a clustered video server in this paper. The scheduling and synchronization techniques for CBR servers are discussed. The techniques for variable-bit-rate (VBR) servers are presented in a different paper [21]. We also present the experiments and performance results. This paper is organized as follows. Section 2 describes the clustered video server architecture. Section 3 presents data partitioning and layout. Scheduling and synchronization for normal play are discussed in Section 4. In Section 5, support for interactive functions are described. Implementation is described in Section 6 and experiment results are presented in Section 7. Section 8 concludes the paper.

2 Video Server Architecture

There are two major types of parallel video servers: shared memory multiprocessors and distributed memory clustered architectures. In a multiprocessor system, there are a set of storage nodes, a set of computing nodes, and a shared memory. The video data is sent to the memory buffer through a high-speed network or bus, and then to the clients. A mass-storage system has presented the capacity of supporting hundreds of requests [9]. However, it is not yet clear that a multiprocessor video server can be scalable. A clustered architecture is easy to scale to hundreds of server nodes. In such a system, a set of storage nodes and a set of delivery nodes are connected by an interconnection network. Data is retrieved from the storage

nodes and sent to the delivery nodes, which send the data to clients. The clustered architecture can be extended to the direct-access architecture, which provides an interface between the storage system and the network [8, 12].

The clustered architecture is shown in Figure 1. There are three kinds of nodes — *storage nodes*, *delivery nodes* and a *control node*. The storage nodes are responsible for storing video data in some storage medium, such as disks. Each storage node deals with its own disk scheduling. The delivery node is responsible for taking requests from clients and forwarding them to the control node for scheduling. Video blocks from storage nodes are buffered in delivery nodes, where video blocks are re-sequenced if necessary, and then sent to clients. Although video streams can bypass delivery nodes and be sent directly to the clients, the delivery nodes have many functions such as buffering and re-ordering video blocks. Thus, the server architecture is made transparent to clients since a client receives video data from a single delivery node that simplifies the system and network management. Admission control, network scheduling, synchronization, and content management of video data reside at the control node. Upon receiving requests from delivery nodes, the control node schedules requests to the next time cycle, if admitted. This centralized control can scale up to at least one hundred nodes. Extra control nodes can be used for more nodes. In this server-push model, unless receiving further notices from clients, the control node keeps updating the current schedule table and sends it to storage nodes for retrieving video blocks. This architecture presents a single-system image to its clients and has been employed by many other servers [7, 11, 10, 22, 23].

Odyssey employs a *flat* architecture where a logical storage node and a logical delivery node are mapped to a single physical node, called a *processing node*. The control node is implemented also on one of the processing nodes. The SCAN algorithm is used for disk scheduling. A buffer for each stream is used to decouple disk access and network transmission so that disk scheduling and network scheduling can be considered separately. Also, time variation in accessing the data in different locations on the disk can be tolerated.

Odyssey uses a Fast Ethernet switch for its interconnection network. Unlike the shared Ethernet, a dedicated link bandwidth is available for each node in the server. To guarantee the real-time constraints, *port contention* has to be addressed. Without scheduling the incoming traffic in advance, multiple packets arriving simultaneously through different incoming links may be forwarded along the same outgoing link. To deal with such output contention, most of modern VLSI switches use buffering where packets failing to be selected for the desired link are buffered inside the switch and go out at a later time. The buffer size is normally not large enough to cope with the worst case of traffic pattern in most of switches where some of packets are still dropped and retransmitted. In Odyssey, our focus is mainly on the design of traffic pattern for a clustered video server to eliminate port contention. Our unique QoS-aware software makes the server

more efficient, more dependable, and more responsive. This suite of software includes conflict-free stream scheduling, server-level synchronization and support of interactive browsing functions.

3 Data Partitioning and Layout

A video file is a sequence of ordered video frames. To facilitate distribution of video data, a video file is partitioned into consecutive video blocks. There are two approaches for partitioning video files — *Constant-Data-Length* (CDL) and *Constant-Time-Length* (CTL). With CDL, a video stream is divided into fixed-size video blocks. Each block has the same data size, but may not have the same play time. On the other hand, with CTL, each block has the same length of play time, but may not have the same data size. Apparently, the CDL partition is suitable for CBR video streams since at each interval the client requires the same size of data. However, many compressed CBR videos are not exactly constant-bit-rate and each CDL block does not play for exactly the same length of time. Although the CDL block is easy to handle, it has some drawbacks. First, some video stream requires more blocks than others, making scheduling more difficult. Second, interactive browsing functions that must drop some frames are very hard to implement with CDL. Taking these issues into consideration, we selected the CTL partition.

With CTL partition, each video block contains a number of video frames. In MPEG [24], several frames can be grouped together so that the frames that depend on each other are confined into the same group as a *group of pictures*. It turns out that such a group can define a natural boundary for partitioning. Therefore, a video block can consist of one or more groups of pictures. The size of video blocks can be determined by optimal I/O access time without loss of parallelism.

Video blocks are distributed to the storage nodes in order to 1) exploit parallelism and hence increase the bandwidth provided for video stream retrieval, 2) maintain load balance over the storage nodes, and 3) reduce the capacity requirement of each single storage node and hence construct a scalable on-demand storage system. In order to serve thousands of clients simultaneously, we can evenly distribute video blocks over N storage nodes in a round-robin fashion, so called *wide striping*. When video blocks are distributed to a subset of nodes, it is called *short striping*. In this paper, we assume wide striping. The results obtained here can be easily extended to short striping.

4 Normal Play

In normal play, data blocks are retrieved from disks in order and sent to clients. For deterministic delivery, data streams are scheduled to time slots to avoid possible port contention.

4.1 Time partitioning

Accessing a video block is accomplished by a thread. The scheduling module must be integrated with real time capability to handle these threads with soft-deadline constraints. When more than one thread accesses the same storage node at the same time, some threads must wait. For this reason, a conventional storage server is typically light-loaded in order to maintain a certain level of performance guarantee. Such a system usually leaves sufficient room for unexpected bursts, resulting in an inefficient use of resources.

When a storage node is dedicated to the retrieval of video streams, the access time and access duration of video blocks can be known in advance. Furthermore, the time of transmitting video blocks in an interconnection network, that is dedicated to video stream transmission, is predictable. To take advantage of this knowledge, retrieval of video streams can be prescheduled. If a storage node can retrieve video blocks for m video streams, the storage node can be time-multiplexed by m . For CBR video, all video blocks are about the same size. Thus, the length of time slots can be equal.

With explicit partitioning of the available bandwidth and prescheduling of video streams onto specific time slots, we constructed a more deterministic scheme. This scheme can better utilize the capacity of resources, provide a better quality of service, reduce the buffer size, and save the real-time scheduling cost. Such a scheme is attractive economically.

The constant time-slot partition is described as follows. The starting blocks of video files are uniformly distributed over all disks. Depending on a selected block size s and the base stream (play) rate R , time is partitioned into *time cycles*, where the length of a cycle is $t_c = s/R$. In general, the data transfer rate of a single disk or a disk array can be much higher than base stream rate R . Therefore, in a time cycle, multiple requests can be serviced by a storage node with time multiplexing. Thus the time cycle is divided into *time slots*, where the length of the slot, t_s , is equal to or longer than the time required for retrieving a block from the storage node or transmitting to the delivery node, whichever is larger. The number of slots in a cycle, m , is determined by $m = \lfloor t_c/t_s \rfloor$. The concept of the time-slot might not seem necessary in a single node server, however, it is extremely important in parallel servers to avoid network conflict. A similar model has been used in the Tiger system [12]. This is a very practical model, which has been justified by the experimental results [25], providing a good quality of service.

Improper scheduling of data retrieval may result in various resource conflicts, such as port contention, where two storage nodes are transmitting to a single delivery node, or disk contention, where more than one request retrieves blocks from the same disk at the same time. Such a resource conflict can cause a poor stream throughput. Our conflict-free stream scheduling algorithm eliminates contentions so that high system throughput can be achieved.

Assume that a server consists of N storage nodes and N delivery nodes. An individual request r is handled by a delivery node i , which is responsible for delivering the data blocks retrieved from storage nodes to the client via network during the entire life-time of request r unless a dynamic relocation is required. Since the blocks of a video is consecutively distributed in all N storage nodes, if request r , at time cycle t , retrieves a data block from storage node j , it will retrieve a data block from node $((j + \tau) \bmod N)$ at time cycle $(t + \tau)$, where $\tau = (1, 2, 3, \dots)$. In each time cycle, $(N \times m)$ requests can be scheduled at most.

Video block scheduling can be illustrated by a simple example. Figure 2 shows a schedule, where $N = 4$ and $m = 3$. An entry in the figure shows the request number r_k , the delivery node number i , and the storage node number j . That is, request r_k retrieves a block from storage node j and sends it to delivery node i . A video stream has its access pattern listed horizontally in a row. For example, request r_0 is scheduled to time slot 0 in the first row. For this request, delivery node 0 retrieves a block from storage node 1 in time cycle 0. It then retrieves blocks from storage nodes 2, 3, and 0 in next three cycles. The schedule table is wrapped around. In a time slot, if more than one request needs to retrieve blocks from the same storage node, they compete for the resource. In order to avoid such a conflict, only the requests that access different storage nodes can be scheduled onto the same time slot. Thus, in every time slot, at most N requests can be scheduled, each of which retrieves a block from a different storage node. The *conflict-free schedule* is a schedule where in each time slot, no two video streams request a block from the same storage node. Once the first cycle has a conflict-free schedule, the following cycles will not have conflict.

In this paper, a scheduling algorithm is presented for multiple disks per node. To fully utilize the disk bandwidth, the load must be balanced among the disks.

4.2 Stream scheduling

The conflict-free schedule implies two constraints. The first one is imposed by the link bandwidth of storage nodes, which is measured by the total number of requests retrieving blocks from the storage node j :

$$v_j^t = |\{r \mid S(r, t) = j\}|,$$

where $S(r, t)$ is the request that retrieves blocks from storage node j at time cycle t . Because no more than one request can access the same storage node in each time slot, no more than m requests can access the same storage node in a time cycle. In addition, since each storage node has d disks, no more than m/d requests can access the same disk in a time cycle. The second one is limited by the link bandwidth of delivery nodes. Within a time slot, a delivery node cannot handle more than one stream from different

storage nodes. The total number of requests handled by a delivery node

$$w_i = |\{r \mid D(r) = i\}|$$

is bounded by m , where $D(r)$ is the request handled by delivery node i . These requests may access arbitrary storage nodes, which are not necessarily distinct. These two constraints are summarized in the following Lemma.

Lemma 1: The necessary condition for a conflict-free schedule in a time cycle is

(1) $v_j^t \leq m$ for $j = 0, 1, \dots, N - 1$; and

(2) $w_i \leq m$ for $i = 0, 1, \dots, N - 1$. □

How do we generate a conflict-free schedule for a given set of requests such that $v_j^t = m$ and $w_i = m$ for every j and i ? A Greedy algorithm has been proposed in [7], which schedules requests in their arriving order. Whenever a request arrives, the next available slot that is not in conflict with the existing ones will be assigned to the request. Despite of the simplicity of the algorithm, an optimal conflict-free schedule cannot be achieved. In the following, a systematic approach to generate an optimal conflict-free stream schedule is presented which maximizes the number of video streams to be admitted. This algorithm converts the scheduling problem to a matching problem on bipartite graphs. Then, the perfect matching algorithm can be applied to solve the problem.

Conflict-Free Scheduling (CFS) algorithm.

Construct a bipartite graph G with bipartition (X, Y) , where $X = x_0, x_1, \dots, x_{N-1}$, $Y = y_0, y_1, \dots, y_{N-1}$, and x_i is joined to y_j if and only if delivery node i handles a request that retrieves a block from storage node j . Because each delivery node handles m requests and there are m requests to retrieve blocks from storage node j , the constructed graph is an m -regular bipartite graph. If we can find a perfect matching in G , then we have a set of requests each of which accesses a different storage node from a different delivery node. According to the marriage theorem [26], if G is a k -regular bipartite graph with $k > 0$, then G has a perfect matching. After determining a perfect matching for time slot 0, eliminate the matched edges, the original problem of scheduling Nm requests to m time slots is reduced to a problem of scheduling $N(m - 1)$ requests to $(m - 1)$ time slots. Thus, applying the perfect matching algorithm m times, a schedule for all time slots can be generated. □

The perfect matching algorithm, the Hungarian method, can be found in [27]. Here we give a brief description of the algorithm. Start with an arbitrary matching M . If M is not a perfect matching yet, an M -unsaturated node u is chosen. We search for an M -augmenting path with origin u to construct a larger matching. This procedure is repeated until a perfect matching is found.

An example of the conflict-free algorithm is shown in Figure 3. After three iterations, the resultant

schedule is shown in Figure 3(d).

This algorithm also can be used for the situation of less than Nm requests. When $v_j^t < m$ or $w_i < m$, dummy requests can be inserted. This algorithm should be called when a request arrives while there is no conflict-free slot. When multiple requests arrive simultaneously, these requests are scheduled to empty slots one by one. When a conflict occurs, this algorithm is called to scheduling all the requests instead of only a single request.

The round-robin placement policy could introduce a convoy effect when one node becomes overloaded. That is, the overloaded condition will shift from one node to the next. This convoy effect can be avoided by a random placement [22]. Our conflict-free stream scheduling algorithm completely prevents the convoy effect because this QoS-aware policy eliminates overloaded conditions.

In addition to $v_j^t \leq m$, the number of requests accessing a particular disk k , $v_{j,k}^t$ cannot be more than m/d . When this condition is violated, only m/d requests can be fulfilled by the disk. Other $v_{j,k}^t - m/d$ requests can still be scheduled, but must be delayed for a few time cycles. A *peg-and-hole* algorithm [28] can be used to minimize the delay. A request relocation algorithm [28] can relocate requests when $w_i > m$ for some i .

A new request can be admitted if and only if it can be scheduled without conflict. It could be scheduled to a time slot immediately upon arrival at a delivery node if there is a conflict-free time slot available. Otherwise, the new request must be delayed, or other video streams need to be rescheduled to accommodate the new request. The admission control algorithm is shown in Figure 4. Let T' be the current total number of video streams being serviced. When one or more requests arrive, the condition $T \leq Nm$ is checked, where $T = T' + r$ and r is the number of newly arrived requests. If the condition holds, the requests are admitted. Otherwise, only $r' = Nm - T'$ requests can be admitted and others must be rejected. For the admitted requests, it tests whether some delivery nodes are overloaded. If so, some requests are relocated. The next step is to check whether some storage nodes are overloaded. If so, some requests are delayed. Finally, it checks whether there are non-conflict time slots for the new requests. If not, the conflict-free stream scheduling algorithm is applied to rearrange video streams.

The admission control algorithm deals with a single request and the multiple requests separately. The following steps are applied to multiple requests arriving simultaneously at the system:

- if for any delivery node i , $w_i > m$, the request relocation algorithm is applied;
- if for any disk k in a storage node j , $v_{j,k}^t > m/d$, the request delay algorithm is applied;
- if any request cannot find an empty conflict-free slot, the conflict-free stream scheduling algorithm

is applied.

In the situation that only a single request arrives at the system, a simple policy can be applied, which is described as follows:

- assume the new request r_k arrives, if $w_i > m$, the new request is transferred to an underloaded node;
- if $v_{j,k}^t > m/d$, the request is to be delayed to its nearest hole;
- if there is a conflict-free time slot for the new request, it is scheduled to the time slot, otherwise the conflict-free scheduling algorithm is applied.

The policy of scheduling multiple requests together can reduce the scheduling overhead and sometime lead to a better schedule.

In this admission control algorithm, the complexity of the request relocation, request delaying, and conflict-free algorithm is $O(N)$, $O(mN)$, and $O(mN \log N)$, respectively. The time to execute the admission control algorithm will be reported in section 7.

4.3 Server-level synchronization

In a switched interconnection network, each processing node has a dedicated link connecting to the switch, which is capable of operating data in full-duplex mode, i.e., each node can send and receive data simultaneously. With careful design of conflict-free stream scheduling, each node can operate concurrently disk access and network transmission within a node and between nodes without resources conflict. These operations can be performed in parallel because each storage node has its own dedicated bandwidth for disk access and network transmission. In order to increase system performance, m ping-pong buffers are used for pipelining data blocks between disk access and network transmission. For each data stream, one buffer slot stores the data block retrieved from disk, while the other is used for transmitting data to the delivery node.

Server-level synchronization is performed to synchronize the data movement. A *cycle synchronization* at the end of each time cycle is necessary to preserve the order of data block transmission. In MARS [8], tightly coupled synchronization between server nodes is performed by customized APICs. In Tiger [12], a separate network connecting all nodes is used to facilitate the synchronization task. Although there is no synchronization problem in a video server with the client-pull model, where a client explicitly sends a request to a particular node of the server for a specific video data block [29], the unbalanced workload between nodes may result in poor system throughput.

Moreover, to guarantee conflict-free data transmission on switched interconnection network, server-level synchronization at the end of each time slot — *slot synchronization* — may be applied. Cycle synchronization ensures the right order of receiving data at clients, while slot synchronization eliminates port contentions. In *Odyssey*, we apply a simple centralized synchronization mechanism to implement a barrier synchronization. That is, each storage node sends a synchronization signal *SYN* to the control node. Once the control node receives *SYNs* from every storage node, it broadcasts an *ACK* signal to storage nodes. The overhead of a barrier synchronization at each slot may overwhelm the benefit of eliminating network contention. This trade-off needs to be justified by experiments.

5 Interactive Functions

Interactive functions offer users more flexible control on the content they are watching. In particular, browsing functions help user to allocate the content to be viewed. We have implemented important interactive functions such as pause, resume, fast-forward, and fast-backward.

5.1 Fast-forward and fast-backward

A simple solution to implementing different *fast-forward* (*fast-backward*) ratios is to reserve a separate encoded video file for each ratio. When switching between different play modes, the corresponding video files will be used. This solution requires extra disk space for the interactive operations and the fast-forward (fast-backward) ratios are limited and fixed. We aim for more efficient and flexible solutions to minimize the system requirement and to provide different fast-forward (fast-backward) ratios. One solution is to retrieve the fast-forward (fast-backward) data from the normal video stream, which can be implemented by block or frame skipping. Experiments showed that, with the block-skipping approach, the client got the impression of watching each scene (block) at regular speed with jumps between scenes, similar to watching a slide projector operating at a high speed. Clients can see the details in each scene so that they are able to locate the position of interest [18]. This approach can solve the bandwidth problem and is easy to implement. We have implemented block-skipping in *Odyssey* with two approaches, prefetching and grouping.

The Prefetching Approach

Block-skipping implements fast-forward by skipping video blocks. When performing an *ff* play of ratio f , a block to be played next is the one after skipping $f - 1$ blocks. To avoid a hot spot, the number of storage nodes and *ff* ratio must be relatively prime. An example is shown in Figure 5(a), where N is 5 and the *ff* ratio is 3.

To fully utilize the allocated slot, upon request of an ff , we do not retrieve the block to be played next if it conflicts with other requests in the same slot. Instead, the block that does not conflict with other requests in the same slot is retrieved. This method is called *prefetching*. Since the retrieved block is not delivered immediately, it is buffered in the delivery node. Figure 5(b) shows the prefetching approach for the example. The delivery node receives blocks in the sequence of (0, 6, 12, 3, 9, 15, ...) and it delivers blocks in the sequence of (0, 3, 6, 9, 12, 15, ...).

The Grouping Approach

Different from the prefetching approach, which changes the retrieval order to avoid conflicts, the grouping approach clusters streams into different groups based on their paces. For example, we may have all normal play streams in one group G_1 with $p = 1$, and all ff streams of the same ff ratio f in another group G_f . Furthermore, ff plays with different ff ratios are in different groups. In this way, all requests in the same group will not conflict with each other as long as they do not conflict in the first time cycle. The system allows a request to change its pace p dynamically, and change its group membership subsequently. Therefore, the key to supporting interactive play is how to handle dynamic reconfiguration of groups.

With multiple groups existing in the system, not every time slots can be fully utilized since the number of requests in a group is not necessarily a multiple of N . Figure 6 shows a case with $N = 3$, $m = 5$, and $K = 3$, where K is the number of groups. The number of time slots allocated to group G_p needs to be varied to support dynamic reconfiguration. Two operations have been defined for changing the membership of a request — *adding* and *removing*. Other two operations have been defined to change the number of time slots allocated to a group — *expanding* and *shrinking*. It has been proved that a request can always change to another group as long as the number of requests in the system is no more than $N \times m - (K - 1) \times (N - 1)$ [30].

Both prefetching and grouping approaches can support a full spectrum of interactive browsing functions to all admitted clients with a guarantee of Quality-of-Service. This is an important feature to be targeted, especially for information search and educational uses.

5.2 Pause and resume

The *pause* and *resume* functions have been implemented in Odyssey. The *pause* function is not difficult to implement. We implement two different pause functions. The first one reserves the bandwidth for the client who makes the *pause* request so that the video stream can resume immediately. The second one releases the bandwidth to other clients. From the server's point of view, it is the same as *stop*; however, it is different from *stop* in that a client record is maintained so that the stream can be resumed from where it

pauses. It may take longer to resume, or is not guaranteed to resume if there is no more bandwidth.

The implementation of *resume* is not simple. It should resume from where the video has been paused. When resuming from *ff* or *fb* play, due to buffer and network transmission delay, it is difficult to start normal play from where the viewer pressed the *play (resume)* button. We use a uniform approach to handle this problem. That is, when the *play* button is pressed, the client feedbacks a block ID of the currently played block or frame. Then the normal play can start from exactly where the viewer wants to see.

6 Implementation

The current implementation of Odyssey consists of three Pentium II 333 PCs, running the RedHat 6.0 Linux operating system. Each node is equipped with 64MB RAM and three UW SCSI disks. These nodes are connected by a CISCO CATAYSTA 2916M Fast Ethernet Switch. Video files are partitioned into video blocks and wide striped into three processing nodes.

The system is implemented with three modules plus a client interface. The three modules are the control node module, storage node module, and delivery node module.

6.1 Control Node Module

The task of the control node is to provide functionality of admission control, scheduling requests, server-level synchronization, schedule table delivery and content query. Such services can be realized as four concurrent threads, which are *Proc_Client*, *Scheduler*, *Sync_Sch* and *Content_Query*, as shown in Figure 7. *Proc_Client* performs admission control. *Scheduler* is responsible for processing and scheduling the queued-up requests according to their play modes. *Sync_Sch* is responsible for synchronizing network transmission among storage nodes followed by broadcasting schedules to storage nodes. *Query_Content* is designed for being responsible for a client's query for a list of video files currently available in the video servers.

6.2 Storage Node Module

The primary function of a storage node is to provide storage space for video files. Recall that to balance the workload and aggregate the I/O bandwidth of storage nodes, a video file is wide striping into all storage nodes. To facilitate the search of a video block, each video file in a storage node is associated with an index file, which contains the block position corresponding to the starting address of the file and block

size. Since this information for a video is indexed in order, the entry point for the index of a block number is self-explanatory. Such an index file is loaded into memory as a fast look-up block directory once the corresponding video file is opened for access.

Due to the diverse characteristics between disk access and network transmission, the ping-pong buffer with the length of two time cycles is applied to decouple them. Since the video data are now retrieved and transmitted at the granularity of a time cycle, this approach also offers a chance to optimize disk access by using round-based scheduling algorithms such as *SCAN*. Each disk in a storage node is associated with one access queue handled by threads for reading data to the buffer.

When surplus network bandwidth occurs, a video block is retrieved from a storage node and delivered to a delivery node residing in the same physical node without real network transmission. Without any optimization, such a data delivery between two processes through a UNIX socket may take three memory copies. To save this system overhead, a shared memory buffer is created to facilitate the data delivery.

The storage node module performs four basic tasks, which are receiving schedule tables, dispatching requests into corresponding disk access queues, retrieving video data from disks into buffer, and conveying them to delivery nodes. In order to ensure the correct order of execution, semaphores for synchronization are applied. These tasks — *Recv_Sch*, *Dispatcher*, *Disk_Read* and *Send_Del* — are implemented as threads, as shown in Figure 8. *Recv_Sch* is responsible for receiving schedule tables from the control node. The main task for *Dispatcher* is to open and close sessions of video files and dispatch disk access operations into corresponding disk access queues. Recall that there is one access queue associated with each disk. *Disk_Read* keeps consuming requests in the queue for disk access until the queue is empty. The major task of *Send_Del* is to deliver data from storage nodes to delivery nodes. There are two possibilities for data delivery, which are through the network or shared memory. If the destined delivery node does not reside in the same physical node with the sending storage node, then the header and the body of a buffered block are transmitted into the network through a specified socket as usual. However, if these two nodes reside in the same physical node, only the header of the video block is sent through the connected socket to indicate the availability of video block in shared memory buffer. In addition, the network transmission is synchronized by sending a *SYN* to the control node and waiting for an *ACK* coming back. The received *ACK* explicitly indicates that the network traffic is cleared to start the transmission of the next video blocks.

6.3 Delivery Node Module

In general, the delivery node provides an access point for clients to access video files in the server. When a new client wants to connect to the video server, it first sends a request to the designated delivery node. This request is forwarded to control node for admission control. Once the request is granted, an open

entry of the *client table* in the delivery is used to register the new client. Information for a client such as client ID, socket ID, play mode, etc., are kept in the client table. In order to avoid network conflict, video blocks transmitted from storage nodes are received according to designated schedules, which are periodically broadcasted by the control node.

In addition, each client is associated with a ping-pong buffer of two video blocks to facilitate the data forwarding. When a client switches into fast forward mode, then additional buffer space of $N - 2$ video blocks obtained from the buffer pool is required to re-sequence the order of video blocks before transmission to the client.

Figure 9 shows the system diagram of the delivery node module. It consists of five concurrent threads — *Recv_Sch*, *Recv_Sto*, *Send_Client*, *Proc_Client* and *Proc_New* — which are discussed as follows.

Recv_Sch is responsible for receiving a schedule at each time cycle. Once received, it is passed to the *Recv_Sto* thread to start receiving video blocks based on the order of request in the schedule so that network conflict can be avoided. *Recv_Sto* is responsible for receiving video blocks transmitted from storage nodes. The header of a video block is received to indicate the incoming body of a video block will be received through regular network socket or available in the special shared memory space. *Send_Client* sends video blocks to clients when they are available in the buffer. Threads *Proc_Client* and *Proc_New* accept requests from clients. *Proc_Client* is responsible for processing requests from existing clients and forwarding them to the control node. *Proc_New* processes only new requests.

A client interface has been developed for this video server. It can be used in the set-top-box for a VoD system. Figure 10 shows the interface. It provides a movie list, makes a connection request, and selects different play modes such as *play*, *pause*, *stop*, *fast-forward* and *fast-backward*. For *ff* and *fb*, different ratios can be selected. Currently available ratios are 4 and 8. A trackbar is provided for fast browsing too.

7 Experiments

With the current Odyssey system, we are able to conduct some experiments. Six MPEG-1 movies are stored, *Toy Story*, *Lost in Space*, *Goldeneye*, *Singin' in the Rain*, *Stepmom* and *Top Gun*. The video files are partitioned into consecutive video blocks, which are evenly distributed to three storage nodes (nine disks). Each video block consists of 15 frames (one group of pictures) representing 0.5-second play time. The base stream rate is 1.4Mbps.

We have implemented also a client-pull video server as a baseline video server for comparison. The client-pull model does not have explicit scheduling. The client is responsible for sending a request for

a specific video block to the video server whenever needed. Without loss of generality, the client sends a request to a particular storage node every half second. An FCFS access queue is implemented in each storage node to line up requests. The storage node then sends the video blocks to clients.

In the following, we present a strength test first before the performance test.

7.1 Strength Test

The strength test measures the maximum capacity of the server. It includes the system utilization, disk bandwidth, network bandwidth, and system throughput.

System utilization

Table I lists the percentage of system overhead and utilization for both slot and cycle synchronization. The barrier synchronization time is the round-trip communication time between the storage node and the control node. The time for each synchronization only depends on the number of nodes and is independent of the system load. The measured barrier time is 0.8 *ms* for three nodes. Since the barrier synchronization time is proportional to $\log N$, the system is scalable to hundreds of nodes. The execution time of the admission control program is negligible in this setting since we only have three nodes. The admission control spends less than 0.1 *ms* per cycle for 64 nodes and the system should be able to scale to hundreds of nodes when cycle synchronization is used. Slot synchronization strictly synchronizes the video block transmission and completely eliminates the conflict between different blocks. Cycle synchronization has small synchronization overhead but the conflict between blocks causes some system idle. Since this is a synchronous system, the time between synchronization is used purely for video transmission.

	Slot synchronization	Cycle synchronization
Synchronization	12%	0.16%
Admission control	negligible	negligible
System idle	0.9%	4.2%
System utilization	87%	95%

Table I: Analysis of System Overhead and Utilization.

Disk bandwidth

For the SCSI disk used in our system, we have measured the disk bandwidth for various data block sizes with the SCAN disk scheduling algorithm, as shown in Table II. In general, a large block size results in higher disk bandwidth. However, the delay for initiating a video stream and interactive operations is

proportional to the block size. In Odyssey, we selected the block size of 0.5-second play time. The largest block size is 95KB and the smallest block size is 77KB. The block size is 86KB on average, and the disk bandwidth is about 4.5MB/s. There are three disks in each storage node, so the total bandwidth of a storage node is 13.5MB/s, equivalent to about 77 streams. Using four disks did not improve the performance.

Block size (KB)	43	86	172	344
# of Blocks/s	66	51	28	19
Bandwidth (MB/s)	2.90	4.50	5.05	6.89

Table II: Disk Bandwidth vs Block Sizes.

Network bandwidth

We measured the interconnection network bandwidth. For each link, the measured bandwidth is about 10.2MB/s. When a storage node sends data to itself through the shared memory, it sends a small head message only. Thus, the achievable bandwidth can be even higher. The measured bandwidth with utilization of shared memory is shown in Table III. The network bandwidth is affected by the TCP buffer size. The default TCP buffer size is 64KB, and can be set up to 128KB. In most cases, larger TCP buffer results in higher bandwidth. In Table III, measured bandwidth for the two different TCP buffer sizes is listed. As it can be seen from the table, higher bandwidth can be achieved at smaller block sizes. Therefore, we packetized the block for network transmission to maximize the network bandwidth.

Block size (KB)	11	22	43	86	172
TCP buffer=64KB	13.0MB/s	12.8MB/s	12.2MB/s	11.2MB/s	9.9MB/s
TCP buffer=128KB	14.6MB/s	14.6MB/s	14.4MB/s	13.7MB/s	11.0MB/s

Table III: Network Bandwidth.

Maximum throughput

The system throughput of the server is shown in Table IV, where the CFS algorithm is used for the server-push mode. The prefetching approach is applied for interactive functions. We tested the maximum throughput by increasing the number of requests until the delay exceeds a threshold, that is, more than 5% of streams have a delay more than a given time limit. For server-push mode, the delay time is no more than 1 second, and for client-pull mode, it is no more than 1 second and 5 seconds, respectively, as indicated in the table. With conflict-free stream scheduling and slot synchronization, each storage can deliver 69 streams. Three storage node can deliver 207 streams. It is no difference with or without interactive operations since the synchronous scheduling guarantees that each stream has its own slot for normal or

interactive operation. With cycle synchronization, the system throughput is 228 streams with each storage node delivering 76 streams. For client-pull, two delay restrictions are shown in the table. With 5-second delay restriction, it performs better than the one with 1-second delay but requires more client buffering and imposes at least 5-second on response time. The interactive operations cause more conflicts and result in less streams supported. The client-pull server does not perform well as the synchronous scheme due to many network conflicts and the convoy effect.

	# of streams per storage node	# of streams with 3 nodes
Server-push, slot sync. all normal	69	207
Server-push, slot sync. 20% interactive	69	207
Server-push, cycle sync. all normal	76	228
Server-push, cycle sync. 20% interactive	76	228
Client-pull, 1-second delay, all normal	42	126
Client-pull, 1-second delay, 20% interactive	37	111
Client-pull, 5-second delay, all normal	58	174
Client-pull, 5-second delay, 20% interactive	55	165

Table IV: Number of Clients Supported by Various Video Servers

7.2 Performance Test

The performance test measures the system performance under various system load. The main performance metric is the rejection rate. The delay time and buffer usage are also tested.

Figure 11 shows the rejection rates for client-pull and server-push with cycle synchronization on the three-node server. The ratio of normal operations and interactive operations is 4:1. For server-push, rejection rates of both the Greedy algorithm [7] and conflict-free scheduling (CFS) algorithm are shown. The Greedy algorithm schedules requests in their arriving order. When the waiting time exceeds certain limit, the request is rejected. For the client-pull mode, a request wait for up to five seconds before it is rejected. The client-pull mode does not guarantee continuous play. Even if the initial delay time is less than five seconds, the delay of some blocks may exceed five seconds. For the server-push mode, a request wait for one second before it is rejected. In fact, if a request cannot be accepted in one second, the system is running in its full capacity, and further requests are unlikely to be accepted even if they are waiting for a long time. That is because the requests must wait until a stream releases its slot, and a stream runs for about two hours. Actually, the video length is chosen from a uniform distribution between 100 and 140 minutes with an average of 120 minutes. The full capacity here is defined as 76 stream per node and 228 streams for the three-node server. Thus, 100% load implies an arrival rate of $\frac{228}{120} = 1.9$ requests per minute.

The client-pull mode does not schedule the streams so a request may wait for a long time to start. The Greedy algorithm, though performs better than the client-pull mode, does not schedule streams well and has many conflicts, resulting in a high rejection rate. The CFS algorithm fully utilizes the system capacity. It only rejects requests when the system runs in its full capacity. Only 4.8% requests are rejected when the arrival rate reaches 1.9/minute.

The above performance test is for a three-node server. To fully understand the performance of the server-push mode in a larger configuration, we have conducted a simulation study. The simulation is configured with five parameters: the total number of nodes N , the number of slots per time cycle m , the average video length size L , the load measurement β based on the available capacity, and the duration of simulation. In the following simulation, the average video length is 120 minutes. and the simulation duration is 24 hours.

The request arrival pattern is Poisson distribution with an arrival rate, a , which can be calculated from the following equation:

$$a = \frac{N * m * \beta}{L},$$

where β is the percentage of full load, $N * m$.

The simulation proceeds step by step for each time cycle as follows:

- (1) At each step the number of new requests is calculated according to Poisson distribution with arrival rate a .
- (2) For each new request, randomly generate its delivery node, its starting storage node where the first block of the file is stored, and the file size.
- (3) The admission control policy is applied to determine which requests are to be admitted or rejected, and if admitted, the scheduling algorithm is used to schedule the requests. If a single new request is generated in this step, the algorithms for the single request are employed. If more than one new requests are generated, the algorithms for multiple requests are employed.

At the end of the simulation, statistics obtained include the total number of requests generated, the number of rejected requests, the initial response time, etc.

Figure 12 compares performance of the Greedy algorithm and the CFS algorithm. The number of nodes N is 16, the number of slots m is 80, thus the capacity is $16 * 80 = 1280$ streams. The arrival rate for the 100% load is 10.66/minute. Obviously, the rejection rates of all algorithms become high as load increases. For CFS, the rejection rate remains around 0 until the load becomes 80%, and at load of 80%,

the rejection rate is 0.1%. For 90% load, the rejection rate of Greedy is 12.3% and that of CFS is only 1.2%. The initial delay time is shown in Figure 13. What we measured was only the server-level delay. The end-to-end delay may take a longer time, depending on the traffic and the size of client buffer. With the server-push mode and its admission control α , algorithm QoS guarantee is provided. Once a request is accepted and a stream is scheduled, at every time cycle, a data block will be delivered. The initial delay is between 0.5 and 1 second under a light load. For a heavy load, some requests cannot find an immediate slot to schedule and must be delayed for a few cycles.

Figure 14 shows performance for different values of m , the number of slots in a cycle. When m increases, the rejection rate decreases. The partial reason for this phenomenon is that when each delivery node handles more requests the variation of the number of requests among nodes becomes smaller. Figure 15 shows performance for different number of nodes, N . There is no substantial change in rejection rates when N varies for the Greedy algorithm. But for the CFS algorithm the rejection rate decreases when N increases, since there could be more chances to relocate the newly arrived requests.

Prefetching and grouping approaches guarantee QoS of interactive functions. In the prefetching approach, fast-forward data can be delivered after a predefined delay time which can be minimized by a proper design. The buffer requirement is fixed and can be further reduced by sharing buffer space among streams. In the following simulation, we add one more parameter: the percentage of ff operations in addition to the five parameters in the normal play.

(a) For Prefetching approach

(i) At each step some of the clients are randomly chosen and their ff ratios are changed.

(ii) For each client, data are fetched from the storage node, as per the original schedule. If the data fetched is not needed immediately, it is stored in a buffer. Delay is calculated for each interactive operation.

(b) For Grouping approach

(i) At each step some of the clients are randomly chosen and their ff ratios are changed.

(ii) Due to the change in the ff ratio of the clients, the number of slots required for a particular ff ratio group may increase. If slots are available, more slots are allocated to that group. If no more slot is available, shrinking is applied to other groups until enough slots are available. We limit the number of admitted clients to $N \times m - (K - 1) \times (N - 1)$, where K is the number of groups [30]. Thus, no ff request will be rejected.

The simulation was run with $N = 17$ and $m = 80$. The permitted ff ratios were 1 (normal), 4 and

8. Figure 16 shows the maximum buffer size required in a delivery node under different load, where the percentage of interactive operations is 20%. The buffer requirement increases as the load increases. Figure 17 shows the maximum buffer size required for different numbers of interactive operations, where the load is 90%. As the percentage of interactive operations increases, the buffer required also increases.

The average delay of interactive operations are shown in Figures 18 and 19, where the 100% load is $17 * 80 - 2 * 16 = 1328$ streams. In Figures 18, the percentage of interactive operations is 20%. In Figures 19, the load is 90%. In the prefetching approach, the delay increases with the load and the number of interactive operations. In the grouping approach, the value is relatively low when the load is less than 60%. When the load is more than 60%, there is a steep increase in the average delay. The average delay decreases as the percentage of interactive browsing operations increases beyond 20% since many interactive operations forms more *ff* groups.

8 Conclusions

Odyssey is a CBR video server with stream scheduling, synchronization, and fast-forward functions. The ultimate goal of Odyssey design is its efficiency, functionality, and responsiveness. An admission control with QoS performance guarantee ensures that after a request is accepted, the server will deliver every block of its stream on time for normal play or interactive play. The delay of interactive operations is bounded. A three-node implementation demonstrates its high performance, interactive browsing functions, and low delay. This clustered server configuration can deliver 228 MPEG-1 video streams of a mix of normal play and *ff* play at the cost below \$10,000, resulting in less than \$50 per stream.

Acknowledgments

This research was supported partially by NSF grants CCR-9505300 and CCR-9625784.

References

- [1] E. Chang and A. Zakhor, "Scalable video data placement on parallel disk array," in *Proceedings of storage and retrieval for image and video databases II*, Oct. 1994.
- [2] D. Ghandeharizadeh and L. Ramos, "Continuous retrieval of multimedia data using parallelism," *IEEE Trans. Knowledge Data Engineering*, vol. 5, pp. 658–669, Aug. 1993.

- [3] A. L. N. Reddy and J. Wyllie, "Disk-scheduling in a multimedia I/O system," in *the first ACM Int'l. Conf. on Multimedia*, pp. 225–233, Aug. 1993.
- [4] H. Vin and V. Rangan, "Admission control algorithm for multimedia on-demand servers," in *The third international workshop on network and operating system support for digital audio and video*, pp. 56–69, Nov. 1992.
- [5] J. K. Dey, C. S. Shih, and M. Kumar, "Storage subsystem in a large multimedia server for high-speed network environments," in *IS&T/SPIE symposium on electronic imaging science and technology*, Feb. 1994.
- [6] C. Martin, P. Narayanan, B. Ozden, R. Rastogi, and A. Silberschatz, *Multimedia Information Storage and Management*, ch. Ch.5 The Fellini Multimedia Storage Server. Kluwer Academic Publishers, 1996.
- [7] A. L. N. Reddy, "Scheduling and data distribution in a multiprocessor video server," in *the second IEEE Int'l. Conf. on Multimedia Computing and Systems*, 1995.
- [8] M. M. Buddhikot, G. M. Parulkar, and J. R. C. Jr., "Design of a large scale multimedia storage sever," *Journal of Computer Networks and ISDN Systems*, pp. 504–524, Dec. 1994.
- [9] J. Hsieh, M. Lin, J. C. L. Liu, D. Du, and T. Ruwart, "Performance of a mass storage system for video-on-demand," *Journal of Parallel and Distributed Computing*, vol. 30, pp. 147–167, Nov. 1995.
- [10] R. Tewari, D. Dias, R. Mukherjee, and H. Vin, "Design and performance tradeoffs in clustered multimedia servers," in *the Third IEEE Int'l Conf. on Multimedia Computing Systems*, June 1996.
- [11] D. Jadav, C. Srinilta, A. Choudhary, and P. B. Berra, "Techniques for scheduling I/O in a high performance multimedia-on-demand server," *Journal of Parallel and Distributed Computing*, vol. 30, pp. 190–203, Nov. 1995.
- [12] W. J. Bolosky, J. S. Barrera, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid, "The Tiger video fileserver," in *NOSSDAV 96*, Apr. 1996.
- [13] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T.-W. Li, "Mitra: A scalable continuous media server," *Multimedia Tools and Applications*, vol. 5, pp. 79–108, July 1997.
- [14] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju, "Staggered striping in multimedia information systems," in *The fifth Int'l conf. on management of data*, May 1994.
- [15] C. Freedman and D. DeWitt, "The SPIFFI scalable video-on-demand server," in *SIGMOD*, June 1995.
- [16] J. Dey-Sircar, J. Salehi, J. Kurose, and D. Towsley, "Providing VCR capabilities in large-scale video servers," in *ACM Multimedia '94*, pp. 25–32, Oct. 1994.
- [17] W. Feng, F. Jahanian, and S. Sechrest, "Providing VCR functionality in a constant quality video-on-demand transportation server," in *ACM Multimedia '96*, pp. 127–135, June 1996.

- [18] M. Chen, D. Kandlur, and P. Yu, "Support for fully interactive playout in a disk-array-based video server," in *ACM Multimedia '94*, Oct. 1994.
- [19] M. Buddhikot and G. Parulkar, "Efficient data layout, scheduling and playout control in MARS," in *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1995.
- [20] A. L. N. Reddy, "Improving latency in an interactive video server," in *SPIE Multimedia Computing and Network Conference*, 1997.
- [21] C. S. Lin, M. Wu, and W. Shu, "Transmitting variable-bit-rate videos on clustered vod systems," in *IEEE International Conference on Multimedia and Expo (ICME2000)*, July 2000.
- [22] R. Tewari, R. Mukherjee, and D. Dias, "Real-time issues for clustered multimedia servers," Tech. Rep. RC20020, IBM Research Report, June 1995.
- [23] R. Haskin and F. Schmuck, "The Tiger Shark file system," in *IEEE 1996 Spring COMPCON*, Feb. 1996.
- [24] D. L. Gall, "A video compression standard for multimedia applications," *Communications of ACM*, vol. 34, pp. 46–58, Apr. 1991.
- [25] C. S. Lin, W. Shu, and M. Wu, "Performance study of synchronization schemes on parallel CBR video servers," in *ACM Multimedia '99, Vol.2*, Nov. 1999.
- [26] P. Hall, "On representatives of subsets," *J. London Math. Soc.*, vol. 10, pp. 26–30, 1935.
- [27] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Macmillian Press, 1976.
- [28] M. Wu and W. Shu, "Optimal scheduling for parallel CBR video servers," *Multimedia Tools and Applications*, May 2001.
- [29] Y. Lee and P. Wong, "A server array approach for video-on-demand service on local area networks," in *IEEE INFOCOM '96*, pp. 1a.4.1–1a.4.8, Mar. 1996.
- [30] M. Wu and W. Shu, "Efficient support for interactive browsing operations in clustered CBR video servers," *IEEE Trans. on Multimedia*, Mar. 2002.

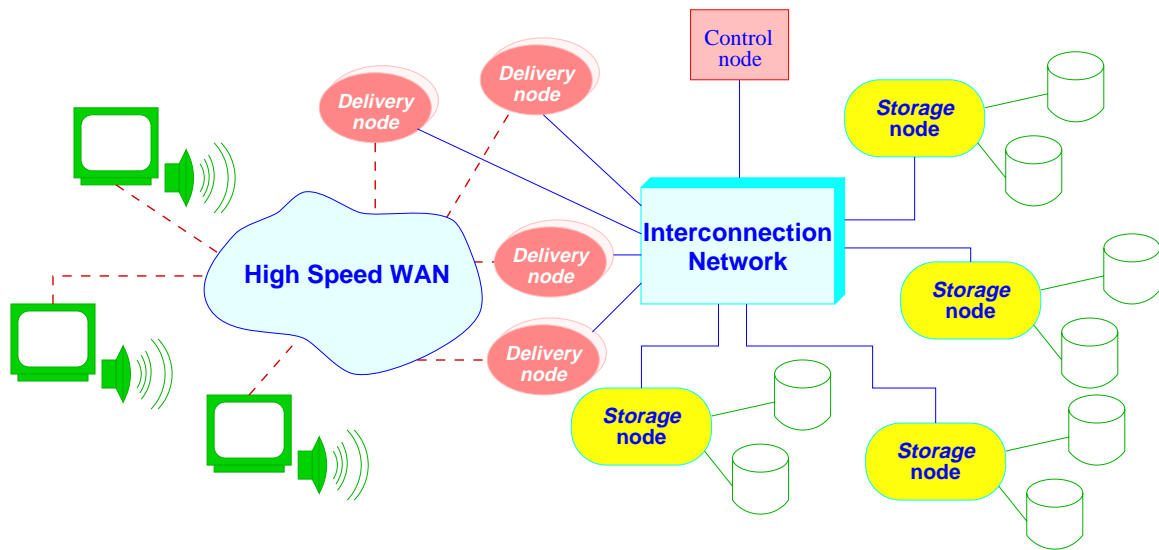


Figure 1: Diagram of a Clustered Video Server.



	time cycle 0			time cycle 1			time cycle 2			time cycle 3		
	slot 0	slot 1	slot 2	slot 0	slot 1	slot 2	slot 0	slot 1	slot 2	slot 0	slot 1	slot 2
node 0	$\begin{matrix} 0 \\ r_0 \\ 1 \end{matrix}$	$\begin{matrix} 0 \\ r_4 \\ 2 \end{matrix}$	$\begin{matrix} 0 \\ r_8 \\ 3 \end{matrix}$	$\begin{matrix} 0 \\ r_0 \\ 2 \end{matrix}$	$\begin{matrix} 0 \\ r_4 \\ 3 \end{matrix}$	$\begin{matrix} 0 \\ r_8 \\ 0 \end{matrix}$	$\begin{matrix} 0 \\ r_0 \\ 3 \end{matrix}$	$\begin{matrix} 0 \\ r_4 \\ 0 \end{matrix}$	$\begin{matrix} 0 \\ r_8 \\ 1 \end{matrix}$	$\begin{matrix} 0 \\ r_0 \\ 0 \end{matrix}$	$\begin{matrix} 0 \\ r_4 \\ 1 \end{matrix}$	$\begin{matrix} 0 \\ r_8 \\ 2 \end{matrix}$
node 1	$\begin{matrix} 1 \\ r_1 \\ 0 \end{matrix}$	$\begin{matrix} 1 \\ r_5 \\ 0 \end{matrix}$	$\begin{matrix} 1 \\ r_9 \\ 2 \end{matrix}$	$\begin{matrix} 1 \\ r_1 \\ 1 \end{matrix}$	$\begin{matrix} 1 \\ r_5 \\ 1 \end{matrix}$	$\begin{matrix} 1 \\ r_9 \\ 3 \end{matrix}$	$\begin{matrix} 1 \\ r_1 \\ 2 \end{matrix}$	$\begin{matrix} 1 \\ r_5 \\ 2 \end{matrix}$	$\begin{matrix} 1 \\ r_9 \\ 0 \end{matrix}$	$\begin{matrix} 1 \\ r_1 \\ 3 \end{matrix}$	$\begin{matrix} 1 \\ r_5 \\ 3 \end{matrix}$	$\begin{matrix} 1 \\ r_9 \\ 1 \end{matrix}$
node 2	$\begin{matrix} 2 \\ r_{10} \\ 2 \end{matrix}$	$\begin{matrix} 2 \\ r_6 \\ 1 \end{matrix}$	$\begin{matrix} 2 \\ r_2 \\ 0 \end{matrix}$	$\begin{matrix} 2 \\ r_{10} \\ 3 \end{matrix}$	$\begin{matrix} 2 \\ r_6 \\ 2 \end{matrix}$	$\begin{matrix} 2 \\ r_2 \\ 1 \end{matrix}$	$\begin{matrix} 2 \\ r_{10} \\ 0 \end{matrix}$	$\begin{matrix} 2 \\ r_6 \\ 3 \end{matrix}$	$\begin{matrix} 2 \\ r_2 \\ 2 \end{matrix}$	$\begin{matrix} 2 \\ r_{10} \\ 1 \end{matrix}$	$\begin{matrix} 2 \\ r_6 \\ 0 \end{matrix}$	$\begin{matrix} 2 \\ r_2 \\ 3 \end{matrix}$
node 3	$\begin{matrix} 3 \\ r_3 \\ 3 \end{matrix}$	$\begin{matrix} 3 \\ r_{11} \\ 3 \end{matrix}$	$\begin{matrix} 3 \\ r_7 \\ 1 \end{matrix}$	$\begin{matrix} 3 \\ r_3 \\ 0 \end{matrix}$	$\begin{matrix} 3 \\ r_{11} \\ 0 \end{matrix}$	$\begin{matrix} 3 \\ r_7 \\ 2 \end{matrix}$	$\begin{matrix} 3 \\ r_3 \\ 1 \end{matrix}$	$\begin{matrix} 3 \\ r_{11} \\ 1 \end{matrix}$	$\begin{matrix} 3 \\ r_7 \\ 3 \end{matrix}$	$\begin{matrix} 3 \\ r_3 \\ 2 \end{matrix}$	$\begin{matrix} 3 \\ r_{11} \\ 2 \end{matrix}$	$\begin{matrix} 3 \\ r_7 \\ 0 \end{matrix}$

Figure 2: Scheduling of Video Blocks for Normal Play.

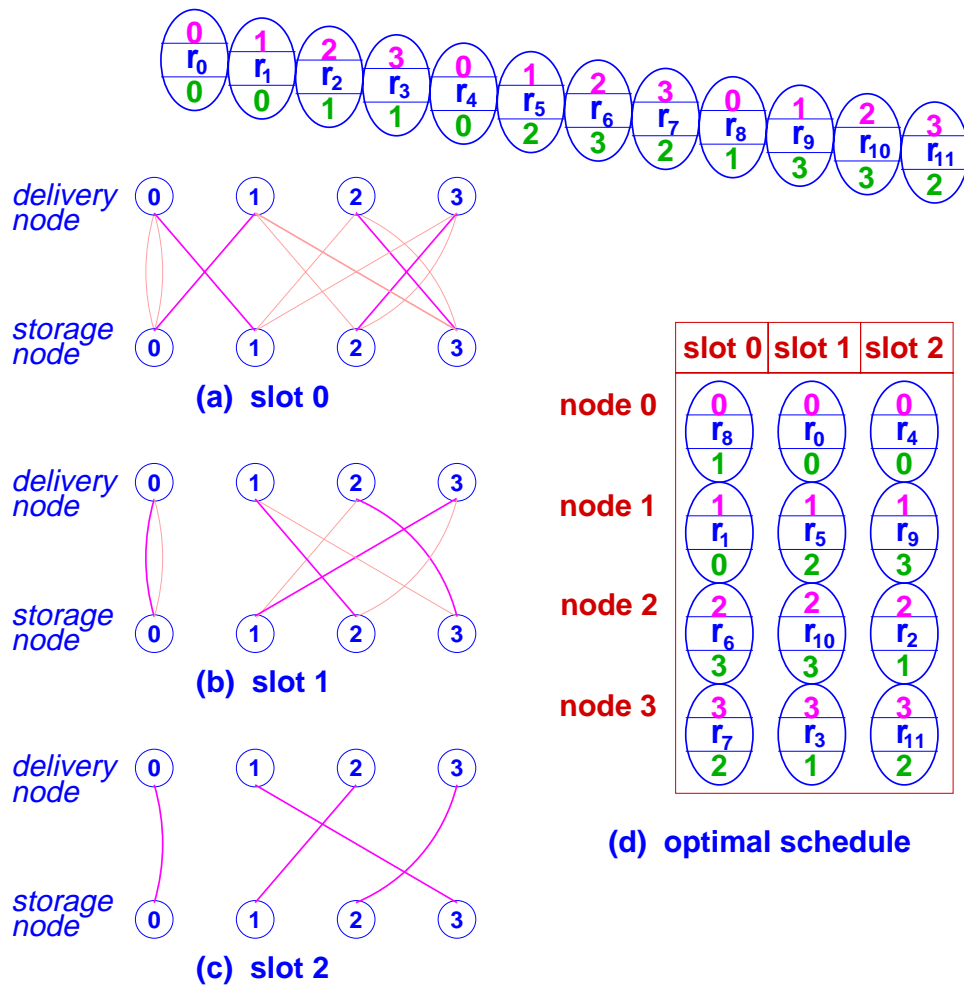


Figure 3: Bipartite Graph for Conflict-free Scheduling

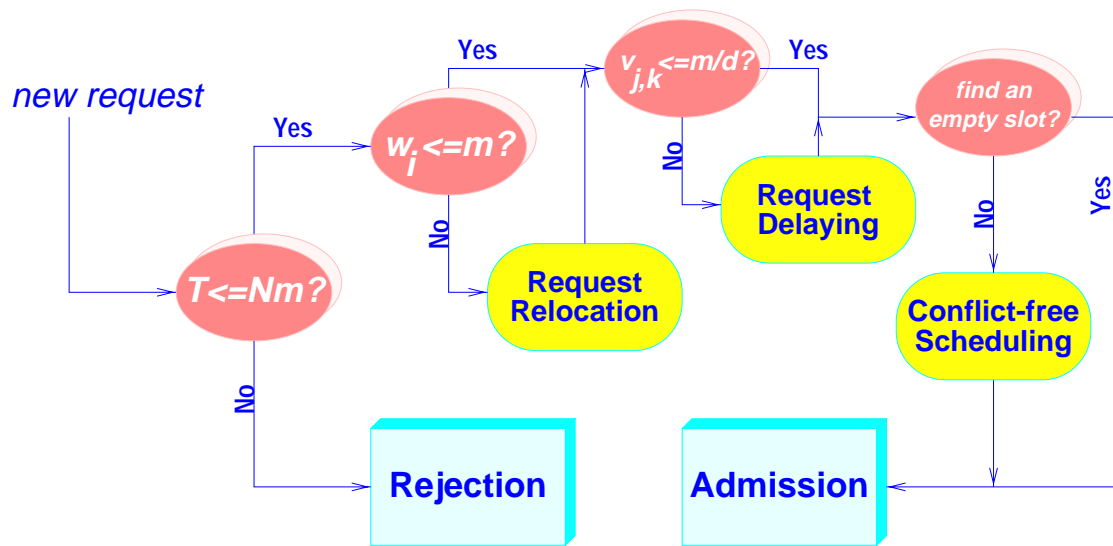


Figure 4: Admission Control Algorithm

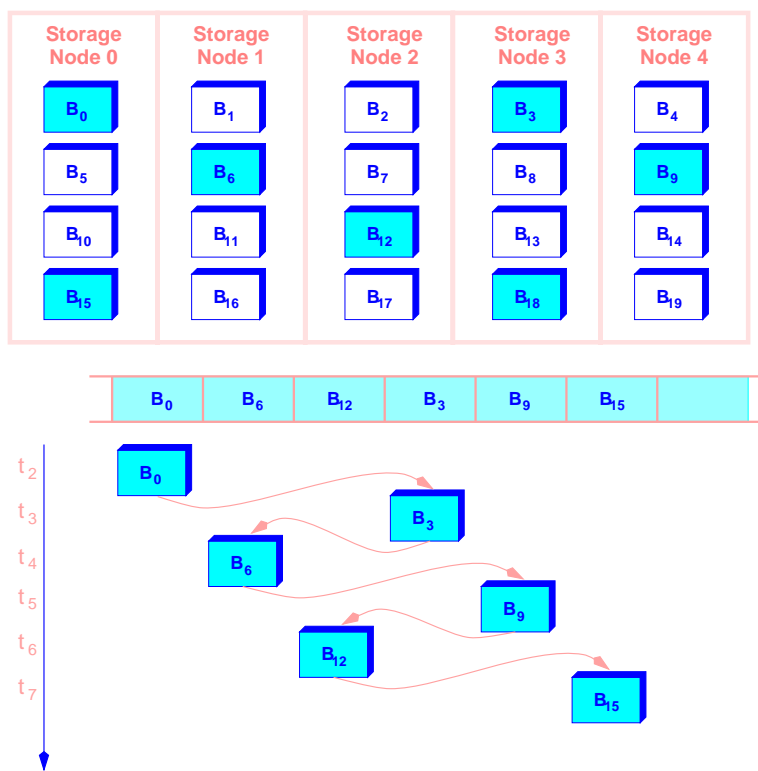


Figure 5: Layout and Access Pattern of Fast Forward for Block-skipping.

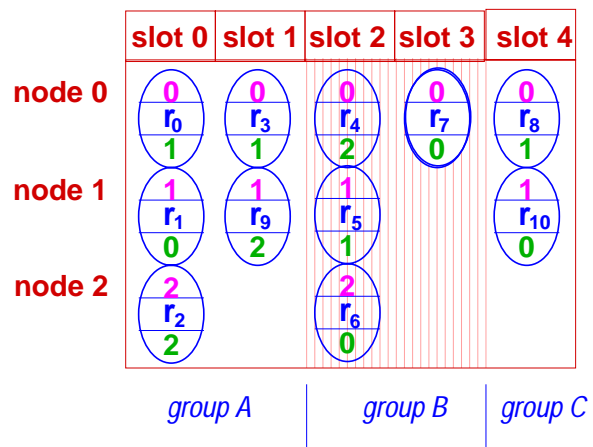


Figure 6: Groups of Requests with Different Paces.

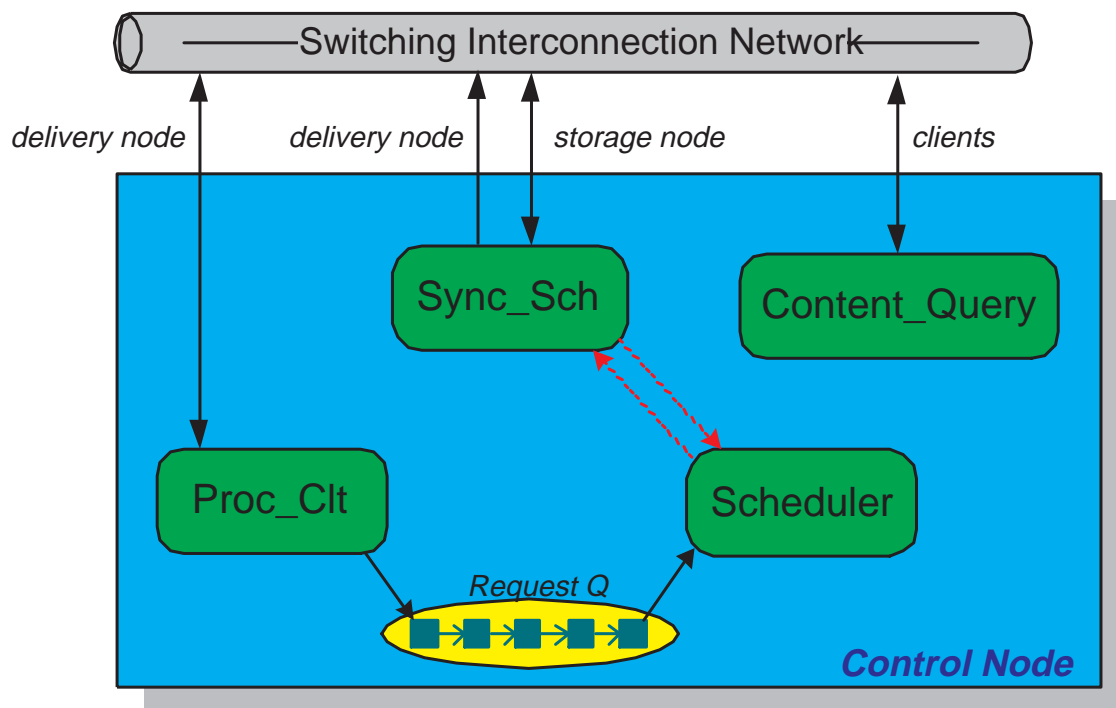


Figure 7: System Diagram for Control Node Module.

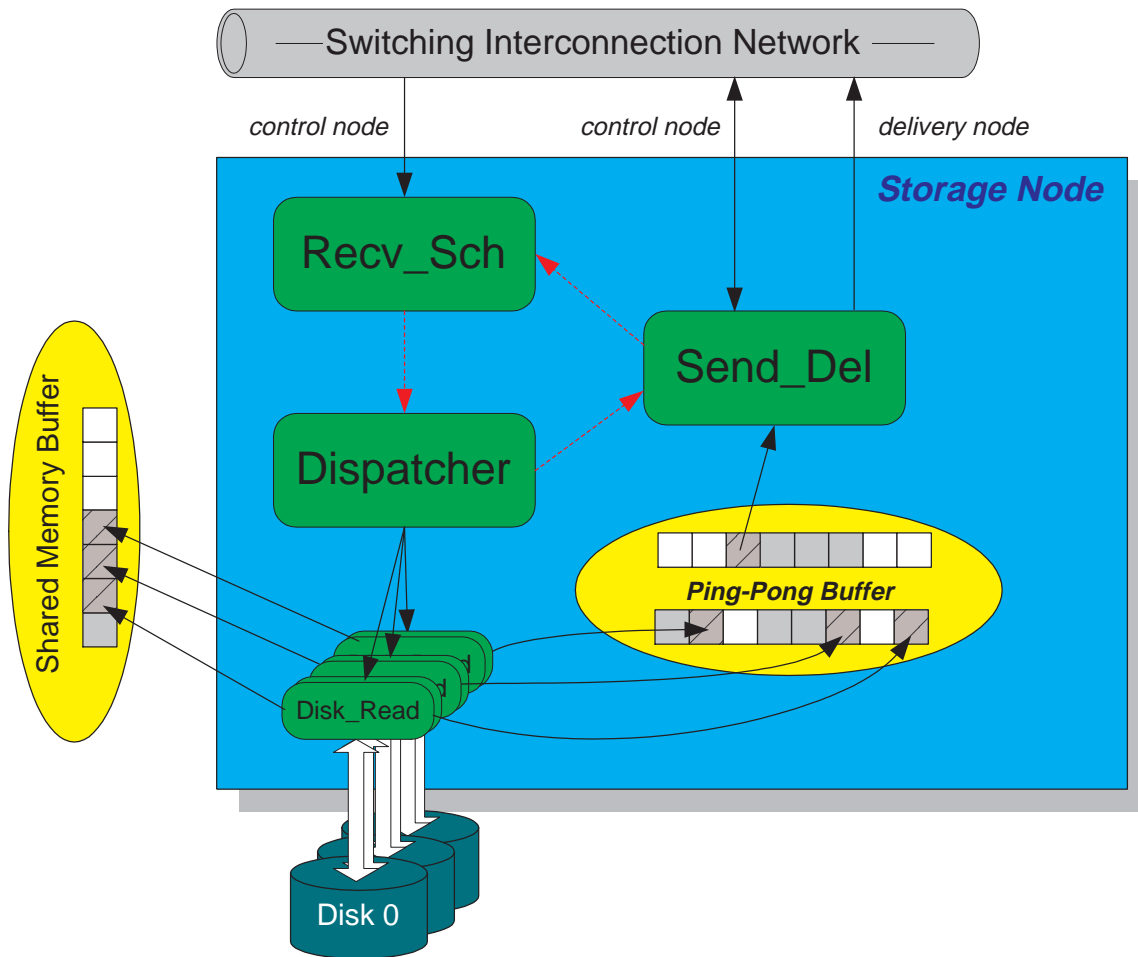


Figure 8: System Diagram for Storage Node Module.

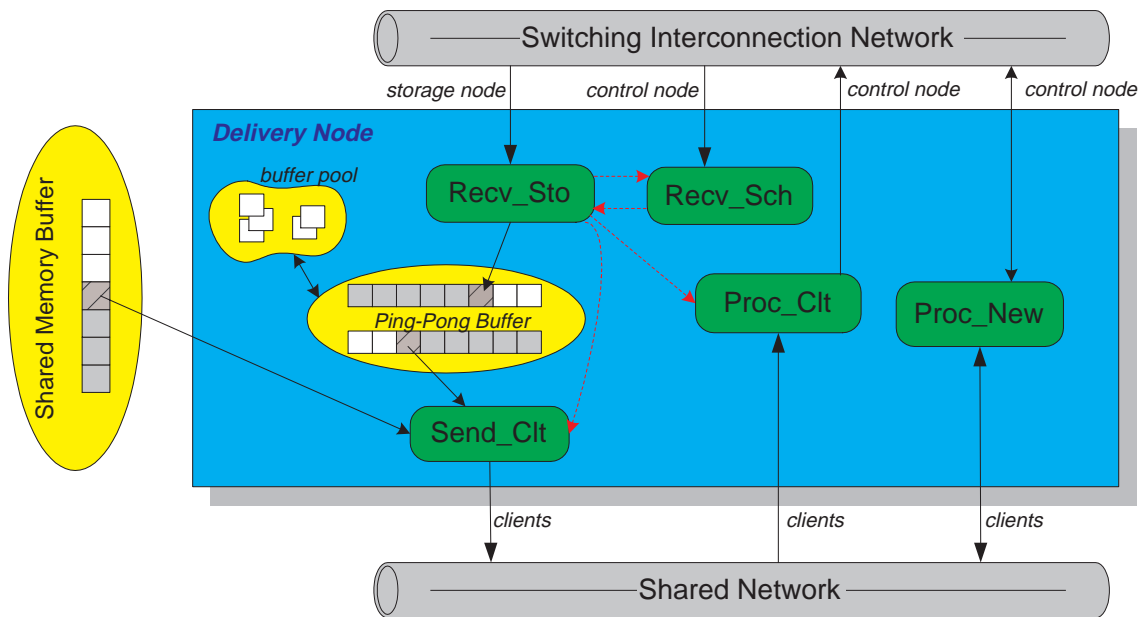


Figure 9: System Diagram for Delivery Node Module.



Figure 10: The Client Interface.

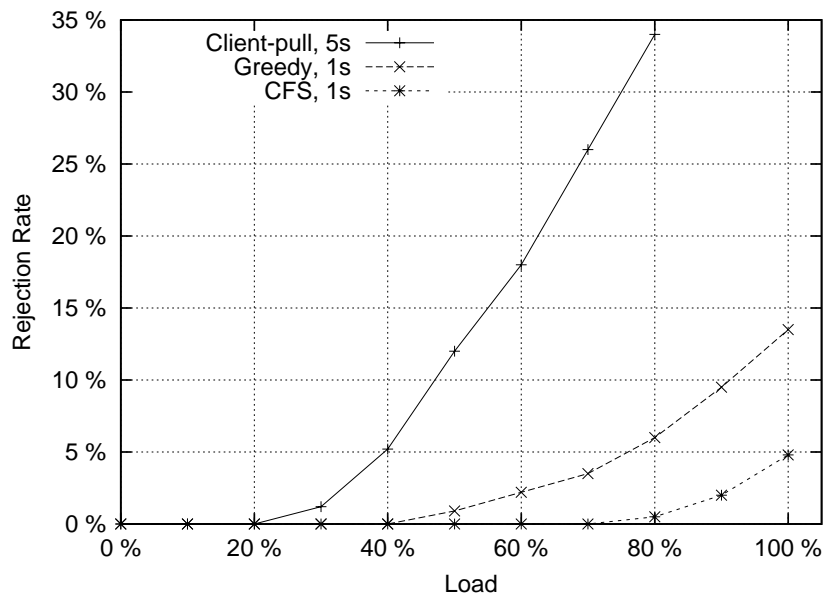


Figure 11: Rejection Rate for Client-pull, Server-push with Greedy, and Server-push with CFS.

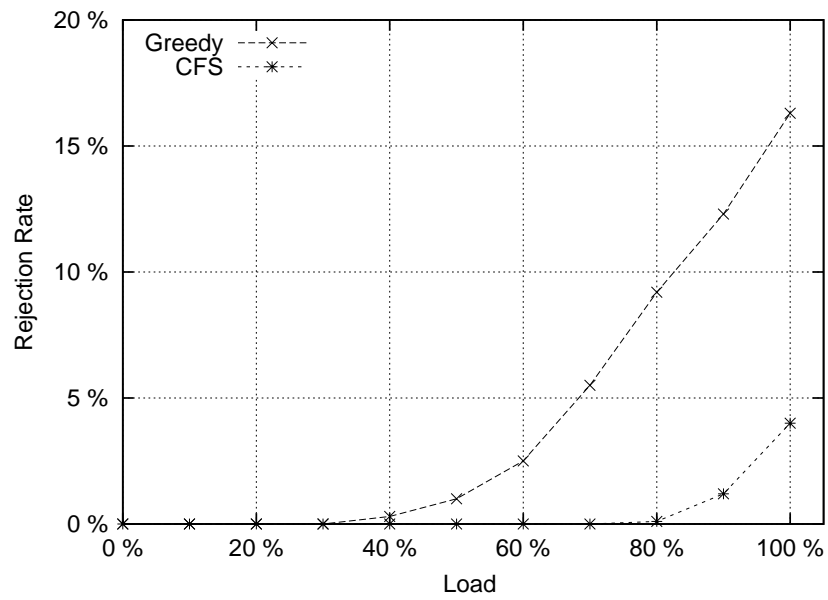


Figure 12: Rejection Rate on 16 Nodes and 80 Slots.

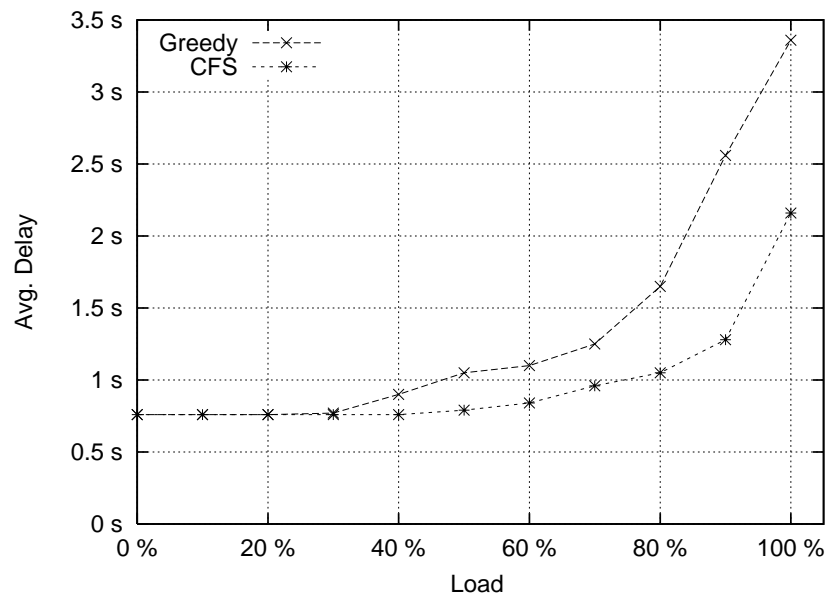


Figure 13: Initial Delay Time on 16 Nodes and 80 Slots.

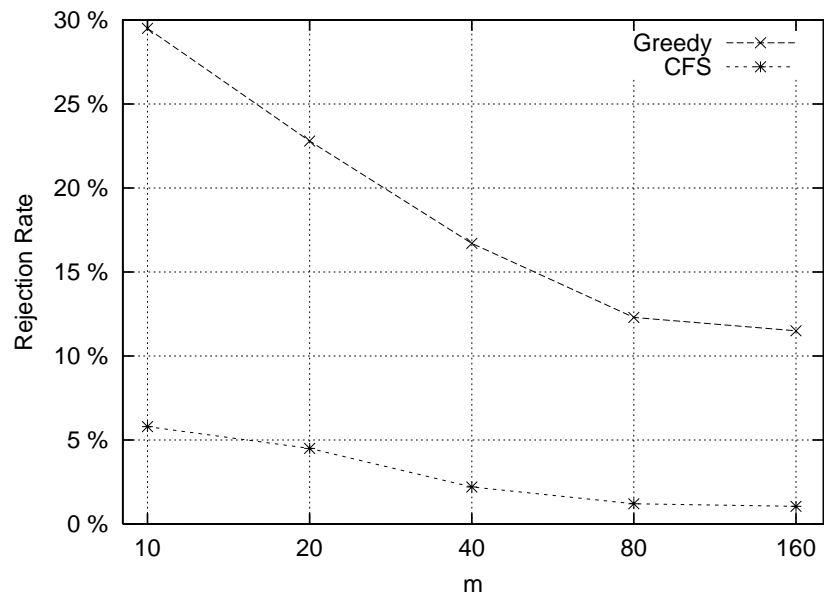


Figure 14: Rejection Rate as Number of Time Slots Varies ($N=16$, $\beta=90\%$).

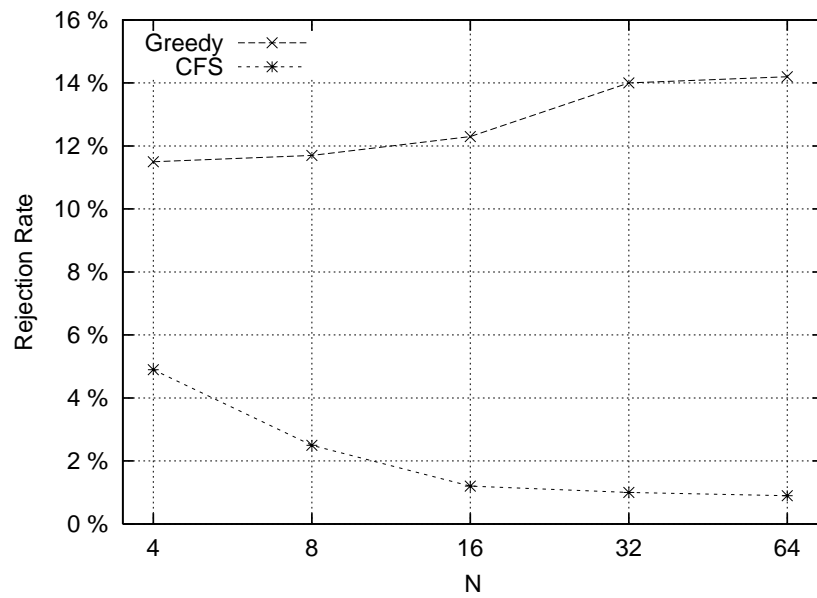


Figure 15: Rejection Rate as Number of Nodes Varies ($m = 80$, $\beta = 90\%$).

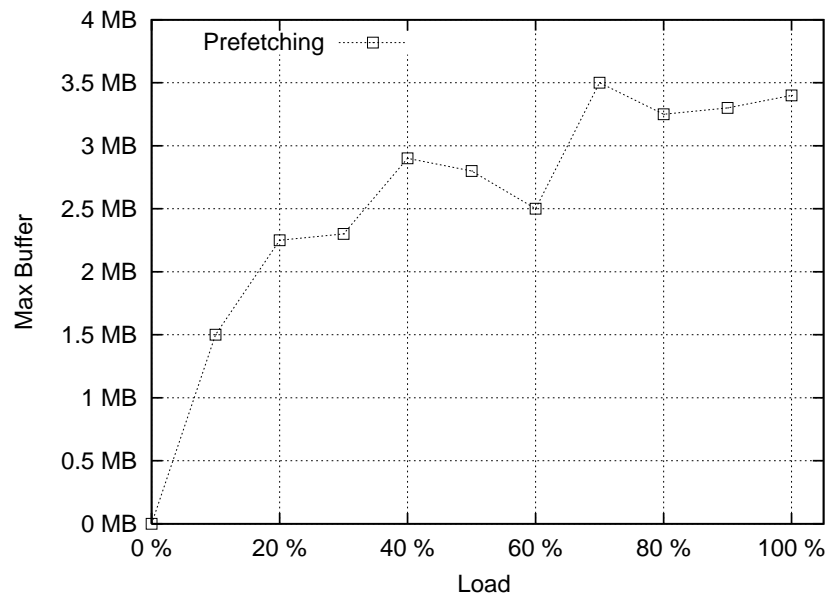


Figure 16: Maximum Buffer as Load Increases in Prefetching Approach.

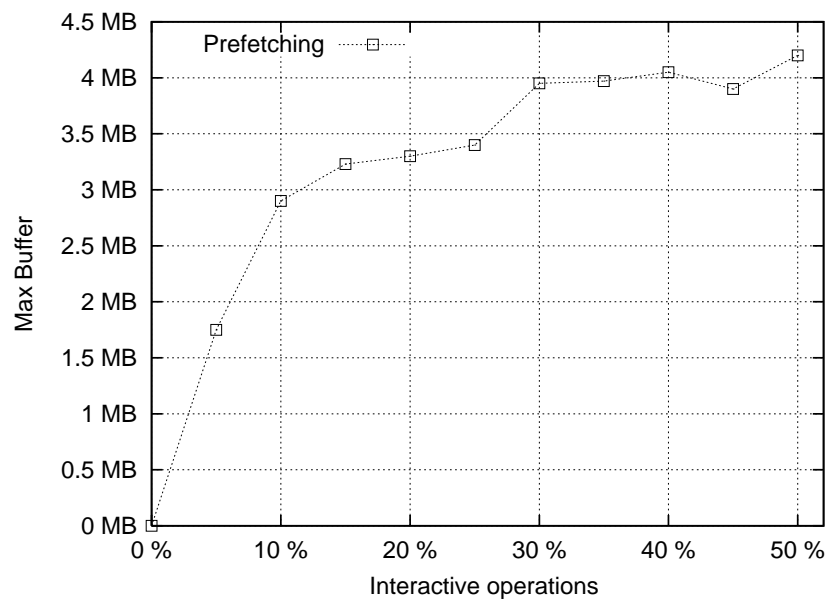


Figure 17: Maximum Buffer as Interactive Operations Increase in Prefetching Approach.

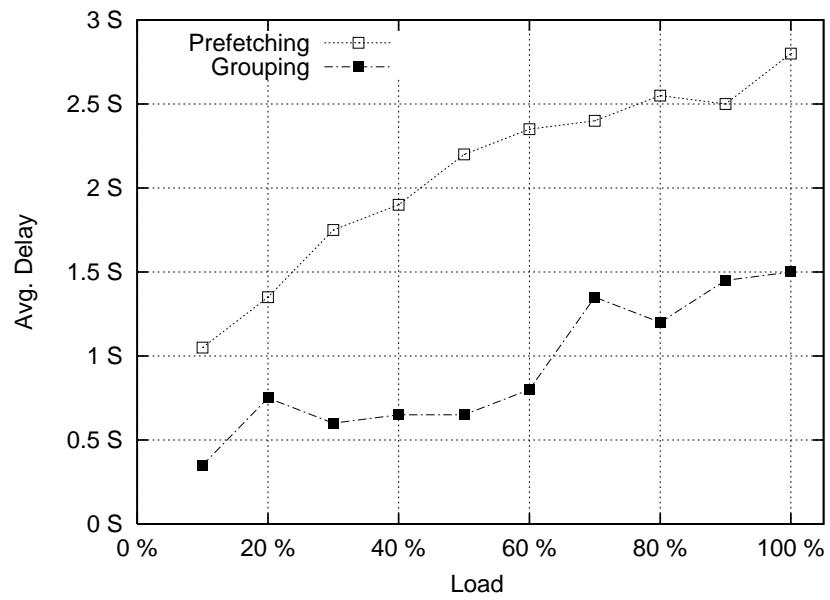


Figure 18: Average Delay as Load Increases in Grouping Approach.

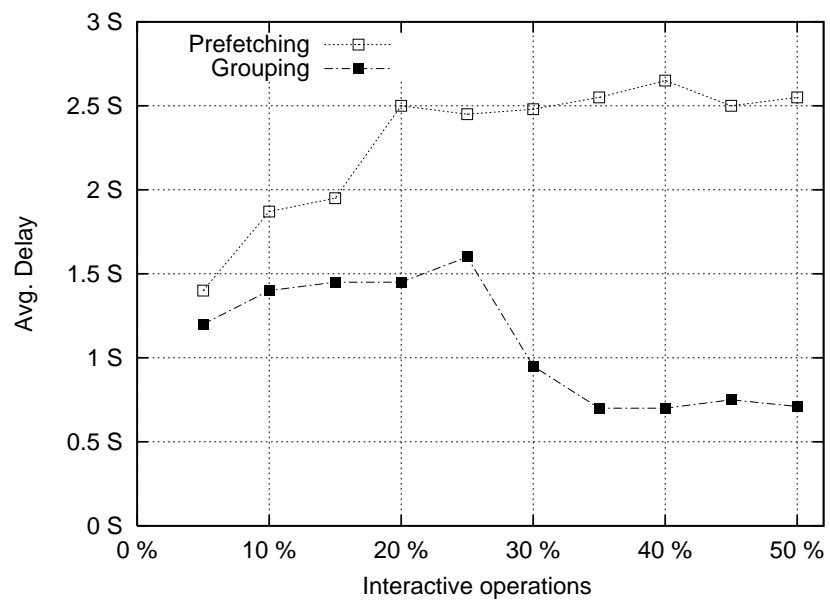


Figure 19: Average Delay as Interactive Operations Increase in Grouping Approach.