Scheduling for Large-Scale Parallel Video Servers

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
wu@cs.buffalo.edu

Abstract— Parallel video servers are necessary for large-scale video-on-demand and other multimedia systems. This paper addresses the scheduling problem of parallel video servers. We discuss scheduling requirements. Optimal algorithms are presented for conflict-free scheduling, delay minimization, load balancing, and admission control.

1 Introduction

There is an increasing demand on capacity of video servers in large-scale video-on-demand systems. Parallel video servers become inevitable to provide large capacity required by the system that can service thousands of concurrent clients.

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 data to be retrieved 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 video streams [9]. However, it is not clear yet that a multiprocessor video server can be scalable. A clustered architecture is easy to scale to thousands 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 storage nodes and sent to delivery nodes which send the data to clients. A number of works describe clustered systems [14, 10, 11]. The clustered architecture can be extended to the direct-access architecture which provides an interface between the storage system and the network. MARS uses an ATM-based interconnect inside the server to connect storage devices to an ATM-based broadband network [3].

In this paper, we evaluate scheduling strategies for the clustered architecture. Figure 1 shows a diagram of clustered video servers. The storage nodes, each with a local disk array, are responsible for storing video data in some storage medium, such as disks. Each storage node deals with its disk scheduling problem separately to provide enough bandwidth. The delivery nodes are responsible for client's requests. On receiving a request from a client, a delivery node will schedule it to a time interval and deliver appropriate data blocks within some time deadline to the client during the playback of a video. The logical storage and delivery nodes can be mapped to different physical nodes of the cluster and are distinct. This configuration is called the "two-tier" architecture in [14]. Also, a node can be both a storage node and a delivery node, called the "flat" architecture.

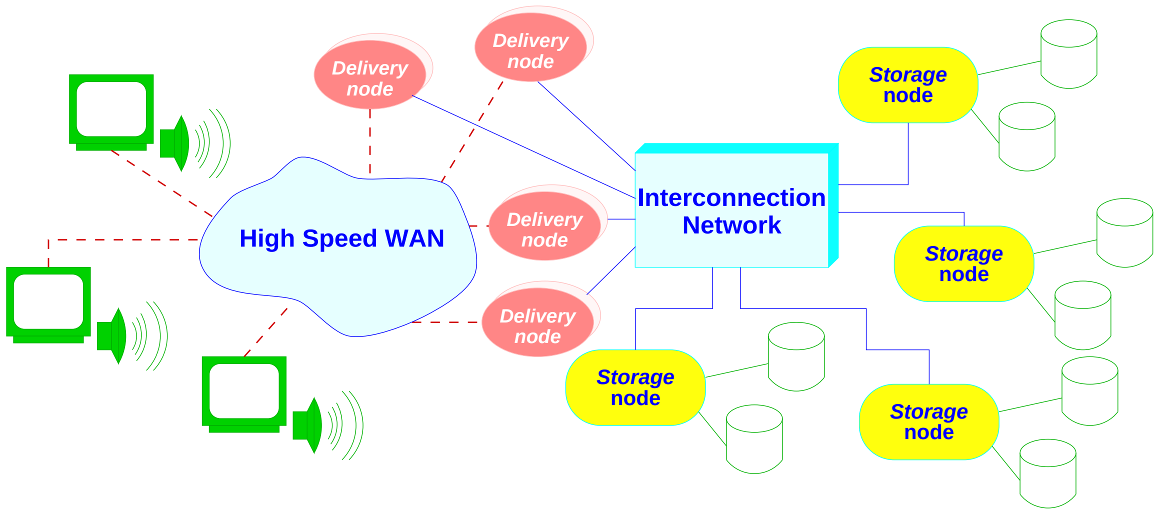

Figure 1: Diagram of Clustered Video Server.

It is the common thought that scheduling policies are critical to performance of video servers. Currently, there are not many scheduling algorithms for parallel video servers, in particular, for clustered video servers. Without a scheduling algorithm, video streams may conflict each other. Therefore, they may be delayed and the quality-of-service cannot be guaranteed. Furthermore, buffers are employed for a smooth delivery due to unpredictable video stream from storage. Video streams that are not well-scheduled may require large buffer space. We need good scheduling algorithms to guarantee the quality-of-service, to efficiently utilize resources and reduce the buffer size, and to increase system throughput.

This paper is organized as follows. Section 2 briefly describes the underlying model of a parallel video server. A conflict-free scheduling algorithm is presented in section 3. Delay minimization is also discussed in this section. In section 4, a load balancing algorithm is presented. Admission control is discussed in section 5. Section 6 gives related work and section 7 discusses the future work.

2 System Architecture and Background

In this section, we model the scheduling problem for video servers. First, we assume that video data is striped across all nodes in a round-robin fashion, so called *wide striping*. Each video starts from different nodes for a balanced load.

Depending on a selected block size s and the base stream rate R, time is divided into *time frames*, where the length of a frame is $t_f = s/R$. The optimal block size is around 256 KBytes to 512 KBytes [11, 14, 9], and an MPEG-2 compressed video rate is about 0.5MBytes/sec. 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 frame, multiple video streams can be serviced by a storage node while the individual stream rate is still preserved. The time frame is thus 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 frame, m, is determined by

$$m = \lfloor t_f/t_s \rfloor.$$

Then t_s is adjusted to t_f/m.

Assume that a large-scale video-on-demand server consists of N storage nodes and N delivery nodes, interconnected by a high-speed network. An individual request r is handled by a delivery node $i = D(r)$, where node i 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 load balancing is required. Since the blocks of a video is consecutively distributed in all N storage nodes, if request r, at time frame t, retrieves a data block from storage node $j = S(r,t)$, it will retrieve a data block from node $(S(r, t+\tau) \ mod \ N)$ at time frame $(t+\tau)$. Thus, in each time frame, at most $(N \times m)$ requests can be scheduled.

Node	Frame 0			Frame 1			Frame 2			Frame 3		
0	1	2	3	2	3	0	3	0	1	0	1	2
1	0	0	2	1	1	3	2	2	0	3	3	1
2	2	1	0	3	2	1	0	3	2	1	0	3
3	3	3	1	0	0	2	1	1	3	2	2	0

Figure 2: A Complete Video Schedule

Video block scheduling can be illustrated by a simple example. Figure 2 shows a schedule, where $N = 4$ and $m = 3$. The starting block of a video stream can be in any storage node. The entries $S(r,t)$ are divided into rows such that all requests of row i are handled by the same delivery node i. An entry shows the storage node number, in a time slot, from where the video blocks is retrieved. A video stream has its access pattern listed horizontally in a row. The blocks of a single stream are separated by m time slots. For example, delivery node 0 schedules the first request that retrieves a block from storage node 1 to the first time slot in time frame 0. The same request retrieves blocks from storage nodes 2, 3, and 0 in next three frames. 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 that retrieves a block from different storage nodes. Once the first frame has a conflict-free schedule, the following frames will not have conflict. Therefore, when discussing the scheduling problem, we can show the first time frame only as long as the access patterns of video streams do not change. Other irregular video streams, such as fast forward, backward, etc. are beyond the scope of this paper.

Now, we define the *conflict-free schedule* as follows.

Definition 1: Conflict-free schedule.

A conflict-free schedule is a schedule that in each time slot, no two video streams request a block from the same storage node. □

A conflict-free schedule imply two constraints. The first one is imposed by the disk retrieving bandwidth of storage nodes which is measured by the total number of requests retrieving blocks from storage node j, at time frame t:

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

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 frame. The second one is limited by the I/O 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 delivery node i

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

is bounded by m. 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 frame is
(1) $v_j^t \leq m$ for $j = 0, 1, ..., N-1$; and
(2) $w_i \leq m$ for $i = 0, 1, ..., N-1$. □

Now, the question is that whether the necessary condition is also a sufficient condition. In the other word, given a set of requests satisfying this necessary condition, does there exist a conflict-free schedule? In the next section, we will prove that Lemma 1 is also a sufficient condition for a conflict-free schedule by giving an algorithm to find out a conflict-free schedule.

3 Conflict-free Scheduling of Video Streams

At this time, we assume that all requests are available at the time of scheduling. We are going to 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 [11], which schedules requests in their arriving order.

Example 1: Assume a set of requests arrive in the order as shown in Figure 3(a). The greedy algorithm schedules these requests as shown in Figures 3(b) to (d). Request r_1 cannot be scheduled to slot 0, so it is scheduled to slot 1; same is r_3. Requests r_4 and r_7 must be scheduled to slot 2. Then, requests r_8, r_{10}, and r_{11} are not able to be scheduled. Thus, only nine requests have been scheduled. $\square$

The greedy algorithm does not guarantee the maximum system performance. Now, we present an optimal algorithm for the conflict-free scheduling problem. 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.

Algorithm 1: Conflict-free scheduling.

Construct a bipartite graph G with bipartition (X, Y), where $X = x_0, x_1, ..., x_{N-1}$, $Y = y_0, y_1, ..., y_{N-1}$, 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 retrieve blocks from storage node j, the constructed graph is an m-regular bipartite graph. For the first time slot, whether there exists a set of requests each of which accesses a different storage node from a different delivery node is equivalent to find a perfect matching in G. According to the marriage theorem [8], 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. $\square$

The perfect matching algorithm, so called the Hungarian method, can be found in [2]. Here

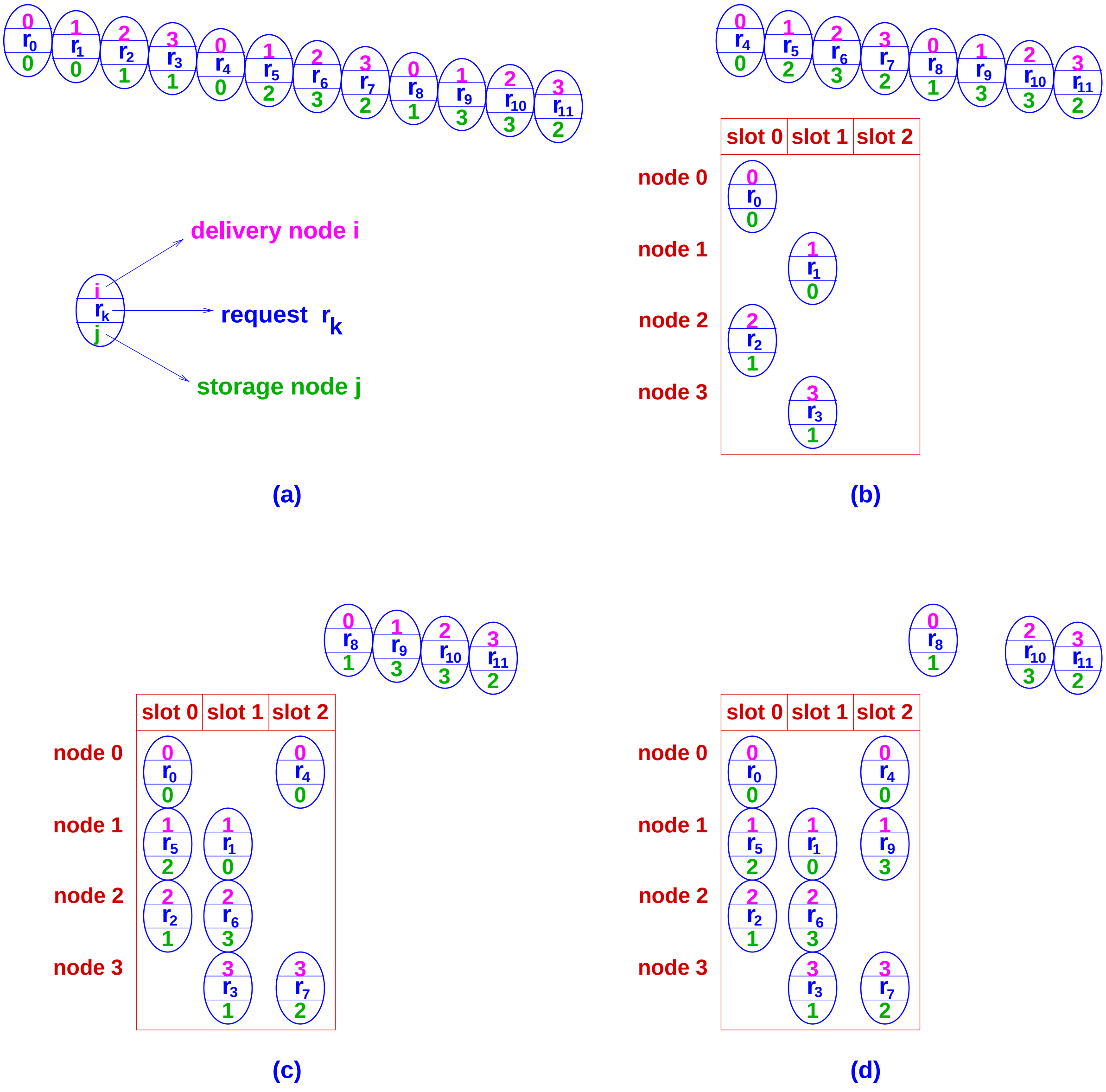

Figure 3: Example of Greedy Scheduling Algorithm

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. The complexity of this algorithm is $N\,log\,N$.

Example 2: An example is shown in Figure 4. It uses the same set of requests in Figure 3. After three iterations, the resultant schedule is shown in Figure 4(d). □

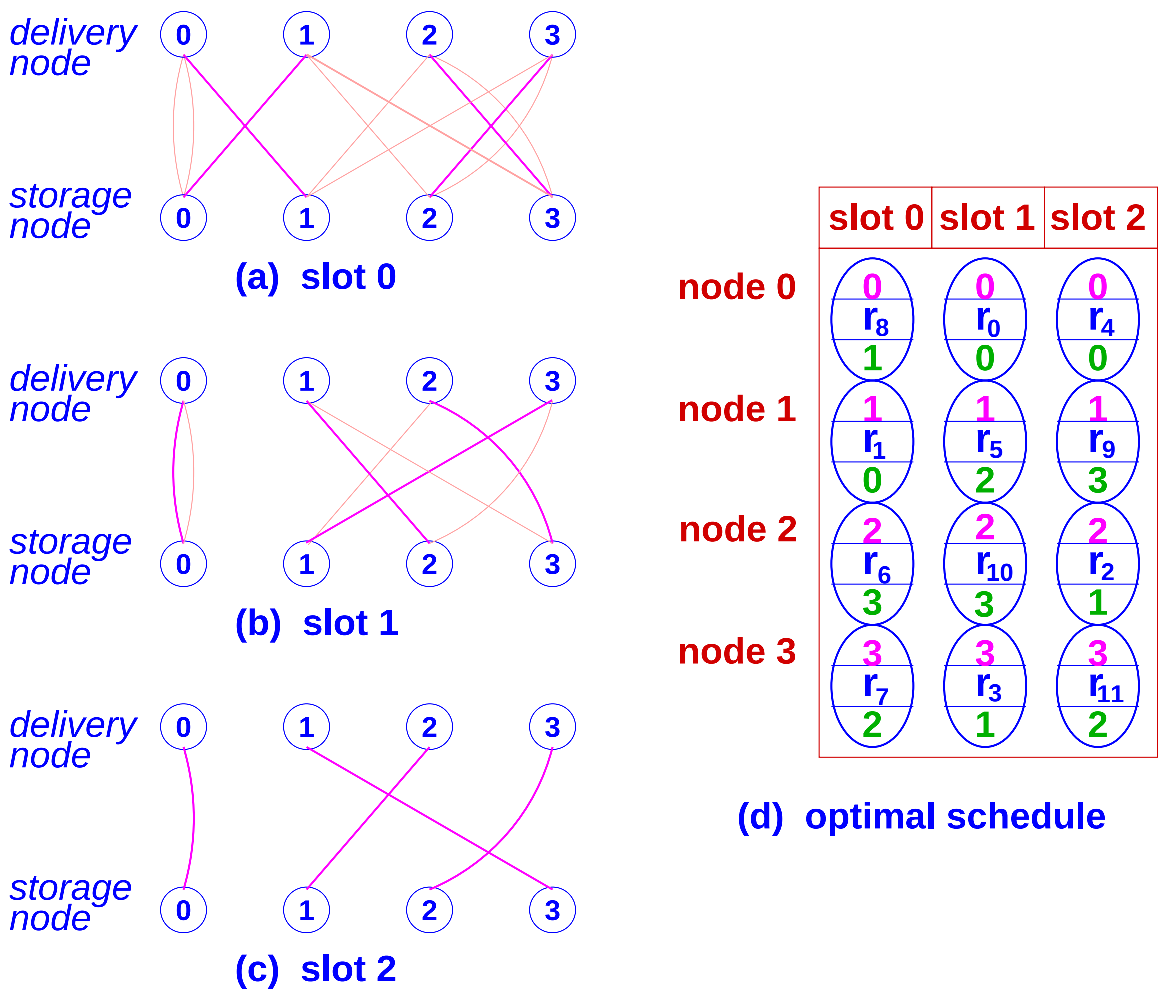

Figure 4: Bipartite Graph for Request Set in Figure 3.

When $v_j^t < m$ or $w_i < m$, dummy requests can be inserted. From this algorithm, it is easy to see that Lemma 1 is also a sufficient condition of conflict-free scheduling.

Request delay

Now consider the situation that the total number of requests is less than or equal to Nm and $w_i \leq m$ for all i, but there exists some j such that $v_j^t > m$. In this case, only m requests can be fulfilled by storage node j, and $(v_j^t - m)$ requests must be delayed. Here, node j is called a "peg." On the other hand, there can be other storage node j where $v_j^t < m$. That leaves $(m - v_j^k)$ empty spaces in node j which is called a "hole." The "peg" requests can be fulfilled by delaying them to "holes". In a large system, the delay can be significant. The problem is how to minimize the delay of these "peg" requests. This problem can be solved with a simple *peg-and-hole* algorithm which is presented as follows.

Algorithm 2: Peg-and-hole.

Let $R(j,k)$ hold the kth requests with $S(r,t) = j$, $v_j^t = \mid \{r \mid S(r,t) = j\} \mid$ and $\beta(r_k) = 0$ for all k

$i_h = N - 1$;
for node $i_p = N - 1$ to 0
  if $(i_p == i_h)$ $i_h = i_h - 1$
  if $v_{i_p} > m$ for each of $(v_{i_p} - m)$ extra requests
    $x = R(i_p, v_{i_p})$; remove request x from node i_p
    while $(v_{i_h} \geq m)$ $i_h = (i_h - 1 + N) \bmod N$; found a hole
    $v_{i_p} = v_{i_p} - 1$;
    $v_{i_h} = v_{i_h} + 1$;
    $R(i_h, v_{i_h}) = x$; transfer it to node i_h
    $\beta(x) = \beta(x) + ((i_p - i_h + N) \bmod N)$;
    $S(x,t) = (S(x,t) - 1 + N) \bmod N$;

□

In this algorithm, each request r_i is represented by $\{S(r_i,t), \beta(r_i)\}$, where $\beta(r_i)$ stands for the units to be delayed on request r_i and initialized to zero. An entry of the two-dimension array $R(j,k)$ indicates the kth request on storage node j. Variables i_p and i_h are indices pointing to the "peg" and "hole" storage nodes, respectively. Basically, a request that cannot be fulfilled in the current time frame is delayed to its nearest unpacked time slot. This algorithm guarantees the minimum delay.

Example 3: Given a set of requests as shown in Figure 5(a), $v_0 = 1$, $v_1 = 5$, $v_2 = 4$, and $v_3 = 2$. Thus storage nodes 1 and 2 are pegs while storage nodes 0 and 3 are holes. Requests r_6 and r_{11} are delayed by two time frames; request r_8 is delayed by one time frame, as given in in Figure 5(b). The resultant schedule is shown in Figure 5(c). □

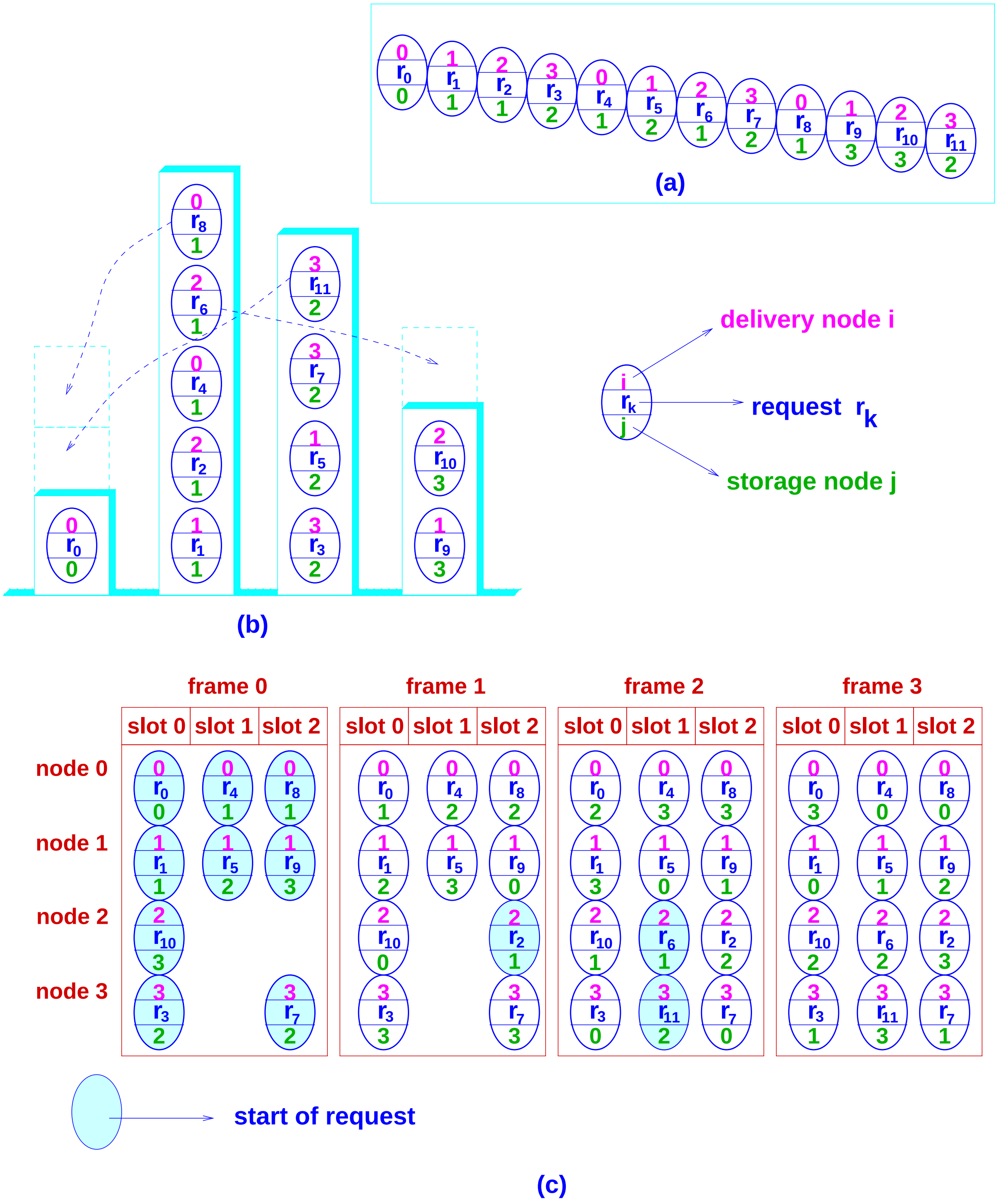

Figure 5: Example for Request Delay

4 Load Balancing

In this section, we consider the situation where the load is unbalanced. The definition of load balance is different from the traditional definition because the video stream scheduling is a real-time problem. As long as every $w_i \leq m$, it is considered as a balanced load. The load needs to be balanced if for some i, $w_i > m$. We assume that the total number of requests is less than or equal to Nm. In the case that the total number of requests is more than Nm, only Nm requests can be fulfilled. Many existing algorithms can be applied to this problem. The objective is to move video streams from those overloaded nodes to the underloaded nodes with minimal cost. RIPS [13], runtime incremental parallel scheduling, is a load-balancing algorithm designed and used to balance independent tasks on large-scale parallel machines. The algorithm can be so modified for parallel video servers that every node that presently handles more than m requests will send excess load to its nearest underloaded nodes. In Figure 6, we present a modified algorithm with assumption that the nodes are arranged in a linear chain. Algorithms for other topologies can be developed similarly.

The first step is to obtain the accumulated number of requests by using the *scan with sum* operation from node $N-1$ to node 0. Each node records a partial sum $T_i = \sum_{j=i}^{N-1} w_j$. In the second step, each node calculates an accumulation quota Q_i. This algorithm can be seen as another "peg-and-hole" algorithm, where the "peg" is the overloaded node and the "hole" is the underloaded node with respect to its quota. The corresponding flows are also calculated. A positive flow means that a flow from node $i-1$ to node i and a negative flow means that a flow from node i to node $i-1$ In the third step, proximities are calculated which are the distance from nearest "hole" nodes. Assume every hole has unlimited capacity, the minimal flow $f_{i-1,i}$ is the result of deliverying every extra work to its nearest hole, in case of a tie, to its left-nearest hole. Hence, there is no flow across any hole node. Therefore, between nodes $i-1$ and i, the resultant minimal flow $f_{i-1,i}$ can be determined only by all nodes from its left-nearest hole to its right-nearest hole, which are bounded by indices from b_i^l to b_i^r. That is, $f_{i-1,i}$ is a summation of possible flows from its left-handed nodes if node b_i^r is their nearest hole and its right-handed nodes if node b_i^l is their nearest hole. In the fourth step, with two flows $y_{i-1,i}$ and $f_{i-1,i}$ available, we can construct an optimal flow $x_{i-1,i}$ such that it is closed to $f_{i-1,i}$ as much as possible, but does not overload any hole node. In the case of $T_0 = Q_0$, the flow $y_{i-1,i}$ is an optimal one. Otherwise, an auxiliary variable g_i is used to record how many streams can be adjusted. Initially, g_0 is the maximum number of streams allowed to be adjusted. Once g_i becomes zero, the rest of optimal flows, $f_{i,i+1}, ..., f_{N-2,N-1}$ must be the same as $y_{i,i+1}, ..., y_{N-2,N-1}$. According to the values of x, the workload is exchanged so that each node has no more than m requests. The number of communication steps in this algorithm is $5N$, where step 1 takes N steps to calculate T_i, step 3 takes $2N$ steps, one for calculating p, one for f, and step 4 takes $2N$ steps, one for

Algorithm 3: Load balancing.

Let w_i be the number of video streams in node i, where $i = 0, 1, ..., N-1$.

1. **Global Reduction:** Perform the *scan* with *sum* operation of w_i:

$$T_i = \sum_{j=i}^{N-1} w_j$$

2. **Quota Calculation:** An accumulation quota and corresponding flow for each node are computed:

$$Q_i = (N-i)m$$

$$g_0 = Q_0 - T_0; \quad for\ i > 0, y_{i-1,i} = Q_i - T_i$$

3. **Proximity Calculation:**

   For each node i, its proximity to the left/right-nearest hole is defined as p_i^l / p_i^r as follows:

$$p_i^l = \min\left(\{i - j \mid 0 \le j \le i \text{ and } w_j < m\} \cup \{\infty\}\right)$$
$$p_i^r = \min\left(\{j - i \mid i \le j < N \text{ and } w_j < m\} \cup \{\infty\}\right)$$

   For each node i, a segment from its left-nearest hole to its right-nearest hole are bounded by indices from b_i^l to b_i^r:

$$b_i^l = \max\left(\{j \mid 0 \le j < i \text{ and } w_j < m\} \cup \{-1\}\right)$$
$$b_i^r = \min\left(\{j \mid i < j < N \text{ and } w_j < m\} \cup \{N\}\right)$$

   Assume every hole has unlimited capacity, the minimal flow $f_{i-1,i}$ is the result of deliverying every extra work to its nearest hole, in case of a tie, to its left-nearest hole.
   For $i = 1, ..., N-1$:

$$f_{i-1,i} = \sum_{b_i^l < j < b_i^r} \begin{cases} m - w_j & \text{if } j \ge i \text{ and } p_j^l \le p_j^r \\ w_j - m & \text{if } j < i \text{ and } p_j^l > p_j^r \\ 0 & \text{otherwise} \end{cases}$$

4. **Load Exchange:**

   For $i = 1, ..., N-1$:

$$g_i = \begin{cases} y_{i-1,i} - f_{i-1,i} & \text{if } g_{i-1} \ge (y_{i-1,i} - f_{i-1,i}) \ge 0 \\ 0 & \text{if } (y_{i-1,i} - f_{i-1,i}) < 0 \\ g_{i-1} & \text{if } (y_{i-1,i} - f_{i-1,i}) > g_{i-1} \end{cases}$$

$$x_i = \begin{cases} f_{i-1,i} & \text{if } g_{i-1} \ge (y_{i-1,i} - f_{i-1,i}) \ge 0 \\ y_{i-1,i} & \text{if } (y_{i-1,i} - f_{i-1,i}) < 0 \\ y_{i-1,i} - g_{i-1} & \text{if } (y_{i-1,i} - f_{i-1,i}) > g_{i-1} \end{cases}$$

   if $i > 0$ and $x_{i-1,i} > 0$, wait to receive $x_{i-1,i}$ tasks from node $i-1$.
   if $i < k-1$ and $x_{i,i+1} < 0$, wait to receive $|x_{i,i+1}|$ tasks from node $i+1$.
   if $i > 0$ and $x_{i-1,i} < 0$, send $|x_{i-1,i}|$ tasks to node $i-1$.
   if $i < k-1$ and $x_{i,i+1} > 0$, send $x_{i,i+1}$ tasks to node $i+1$.

Figure 6: Load-balancing Algorithm for a Chain.

calculating x, one for load exchange. This algorithm minimizes the number of communications and communication distance. It also maximizes locality.

Example 4: Table I and Figure 7 show an example for the load-balancing algorithm, where $m = 4$. T_i, Q_i, and $y_{i-1,i}$ are calculated in steps 1 and 2. In step 3, p_i^l and p_i^r are calculated so that by comparing their values each overloaded node can determine its intention of sending extra load, which is the value of f. For example, node 3 intends to send a request to node 2 so that $f_{2,3} = -1$. Similarly, node 4 intends to send a request to node 5 so that $f_{4,5} = 1$. When $p_i^l = p_i^r$, the node intends to send its extra load to left. In step 4, the value of x is calculated according to the values of y, f and g. Since the value of x must be in range $(y - g, y)$, the final flow could be different from the value of f. For example, node 4 intends to send a request to node 5 according to the value of f, but finally it sends a request to node 3. Otherwise, nodes 5, 6, or 7 would be overloaded. □

Table I: Example for Load Balancing

Node i	0	1	2	3	4	5	6	7
w_i	3	4	0	5	5	1	8	3
T_i	29	26	22	22	17	12	11	3
Q_i	32	28	24	20	16	12	8	4
$y_{i-1,i}$	–	2	2	-2	-1	0	-3	1
p_i^l	0	1	0	1	2	0	1	0
p_i^r	0	1	0	2	1	0	1	0
b_i^l	-1	0	0	2	2	2	5	5
b_i^r	0	2	2	5	5	5	7	7
$f_{i-1,i}$	–	0	0	-1	0	1	-4	0
g_i	3	2	2	0	0	0	0	0
$x_{i-1,i}$	–	0	0	-2	-1	0	-3	1

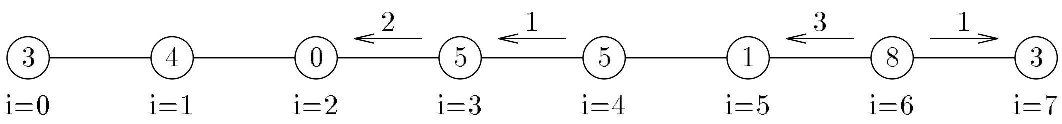

Figure 7: Example for Load Balancing

5 Admission Control

With the conflict-free scheduling algorithm, request delay algorithm, and load-balancing algorithm, we are ready to define the admission control policy for new requests. 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 non-conflict time slot available. Otherwise, the new request must be delayed to start, or other video streams need to be rescheduled to accommodate the new request.

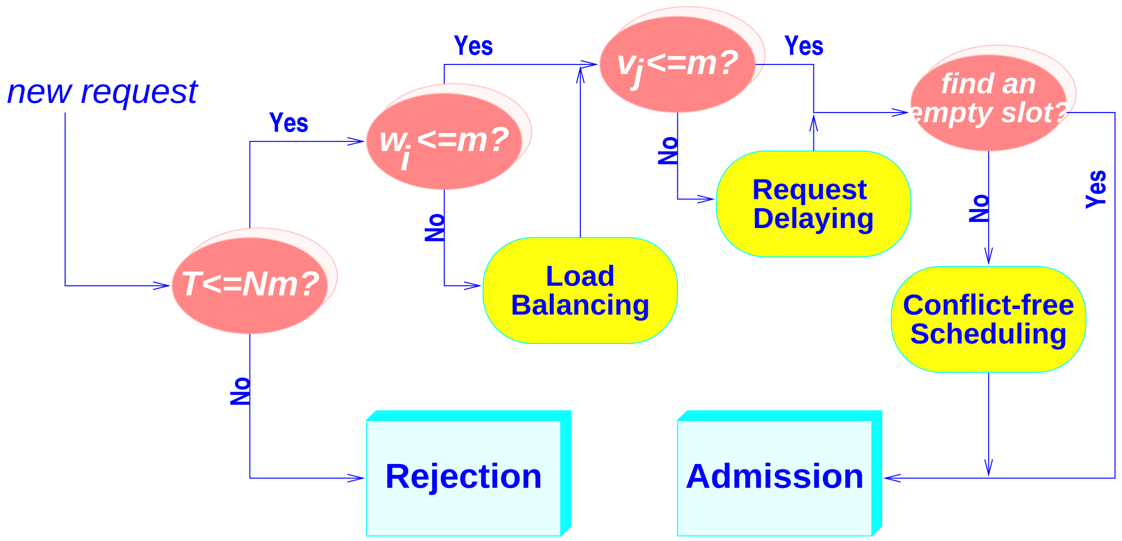

Figure 8: Admission Control Algorithm

The admission control algorithm is shown in Figure 8. Let T' be the current total number of video streams being serviced. When a new request r_k arrives with $D(r_k) = i$ and $S(r_k, t) = j$, the total number of streams to be serviced becomes $T = T' + 1$, if request r_k is admitted. There is a simple way to test schedulability of the new request, that is, comparing T to Nm. If $T > Nm$, request r_k is immediately rejected because the system can service at most Nm requests. If $T \leq Nm$, request r_k can be admitted. The next step is to to test whether delivery node i is overloaded. If $w_i > m$, the load balancing algorithm is applied to find a underloaded node to handle the request. The algorithm can be greatly simplified to handle a single new request. That is, the left-nearest and the right-nearest holes are searched for node i. The new request is then transferred to its nearest hole which is determined by comparing values of p_i^l and p_i^r.

The next step is to check availability of storage node j at time t, where $j = S(r_k, t)$. If $v_j^t > m$, the request delay algorithm is applied. Once again, the algorithm can be simplified for a single new request. That is, the request is to be delayed to its nearest hole. It is then to be checked that if there is a non-conflict time slot for the new request. If so, the request is scheduled to the time slot. If not, the conflict-free scheduling algorithm is applied.

6 Related Work

During the past few years, several research projects investigated techniques for designing video servers. Related issues include disk layout and scheduling strategies, admission control strategies, real-time support, etc. [4, 7, 12, 15, 5]. Most research works assumed single disk storage system. Several research projects investigated the design techniques for parallel video servers [11, 3, 9, 14, 10]. Video data striping schemes have been studied [11, 1, 4, 7]. However, video stream scheduling problem has not been formally addressed. The system resources cannot be fully utilized without a sophisticated scheduling strategy. Not arranging the video streams properly, large buffer space is required. In particular, the prefetch technique requests huge memory space [6]. Our approach aims at accurate scheduling of video streams to maximize system throughput and to minimize the usage of buffers.

7 Conclusion and Future Work

This paper addressed the scheduling problem for parallel video server. A number of algorithms including conflict-free scheduling, request delay, load balancing, and admission control have been presented. Combining these algorithms, we are able to achieve optimal scheduling in distributed memory cluster architectures.

We did not discuss the network conflict problem in this paper. Network traffic scheduling is an important issue to be addressed. For some topologies such as the Omega network and hypercube network, once a set of requests has been scheduled without conflict for a time frame, there is no conflict for other frames. However, for other network topologies, it might be necessary to do scheduling for each individual time frames.

The paper only addressed scheduling problem for regular video data. In some applications, each client may request the retrieval of a variable bit rate encoded media stream. Scheduling this type of requests is much difficult and needs to be addressed in the future work.

Acknowledgments

The authors would like to thank Xin He for his helpful discussion.

References

[1] 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.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Macmillian Press, 1976.

[3] M. M. Buddhikot, G. M. Parulkar, and J. R. Cox Jr. Design of a large scale multimedia storage sever. *Journal of Computer Networks and ISDN Systems*, pages 504–524, December 1994.

[4] E. Chang and A. Zakhor. Scalable video data placement on parallel disk array. In *Proceedings od storage and retrieval for image and video databases II*, October 1994.

[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*, February 1994.

[6] C. Freedman and D. DeWitt. The SPIFFI scalable video-on-demand server. In *SIGMOD*, June 1995.

[7] D. Ghandeharizadeh and L. Ramos. Continuous retrieval of multimedia data using parallelism. *IEEE Trans. Knowledge Data Engineering*, 5(4):658–669, August 1993.

[8] P. Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26–30, 1935.

[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*, 30(1):147–167, November 1995.

[10] 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*, 30(1):190–203, November 1995.

[11] 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.

[12] A. L. N. Reddy and J. Wyllie. Disk-scheduling in a multimedia i/o system. In *the first ACM Int'l. Conf. on Multimedia*, pages 225–233, August 1993.

[13] W. Shu and M. Y. Wu. Runtime Incremental Parallel Scheduling (RIPS) for large-scale parallel computers. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 456–463, February 1995.

[14] 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.

[15] 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*, pages 56–69, November 1992.