# Concurrent Push—A Scheduling Algorithm for Push-Based Parallel Video Servers

Jack Y. B. Lee

***Abstract*—Most existing commercial video servers are designed for a single server. Consequently, the capacity of the system in terms of maximum sustainable concurrent sessions is limited by the performance of the video server hardware. This paper proposes and analyzes the performance of a novel parallel video server architecture where video data are striped across an array of autonomous servers. The architecture allows one to build incrementally scalable video servers without video data replication. The proposed concurrent-push scheduling algorithm allows the system to integrate with quality of service guarantees provided by today's switching networks. In this paper, the striping policy, the service model, and the concurrent-push scheduling algorithm are presented. A system model is constructed to quantify three performance metrics, namely, server buffer requirement, client buffer requirement, and system response time. Results show that a simple extension of the server-push service model does not perform well under the parallel video server architecture. To improve system performance, a novel extension of the grouped sweeping scheme called the asynchronous grouped sweeping scheme (AGSS) is introduced. To further increase the scalability of the architecture, a new subschedule striping scheme (SSS) is introduced. With the proposed AGSS and SSS, our parallel video server architecture can be scaled up to more than 10 000 concurrent users.**

***Index Terms*— Concurrent push, grouped sweeping scheme (GSS), parallel video server, performance analysis, scheduling algorithm, server push, server striping, video on demand.**

## I. Introduction

ONE common architecture shared by most existing video-on-demand (VoD) systems is that they are based on a single server. The video server can range from a standard PC for small-scale systems [1], [2] to massively parallel supercomputers with thousands of processors for large-scale systems [3], [4]. However, the price/performance ratio escalates quickly for high-end hardware, and ultimately the capacity of a single server is still limited. When the demand exceeds the server's capacity, one may need to replicate data to a new server for more capacity, albeit doubling the storage requirement of the system. As high-quality digital video requires a vast amount of storage, this approach is expensive. A second approach is to partition the video titles into disjoint subsets and store each subset in a different video server. This approach does not require extra storage but suffers from load-balancing problems. Empirical studies [5], [6] have shown that video retrievals are highly *skewed*, i.e., some videos are more popular than others. Furthermore, the skewness changes with time (e.g., when most users have seen the video). Hence, under partition, some video servers might be overloaded by a popular title even though other servers might be underutilized.

In this paper, we study a parallel server architecture for designing scalable VoD systems. Unlike replication, we use striping to achieve load sharing across multiple servers without increasing the storage requirement. Furthermore, by striping using a small unit size, the system is insensitive to skewness in video retrievals. This architecture allows one to incrementally scale up the system capacity to more concurrent users by adding (rather than replacing) more servers and redistributing (rather than duplicating) video data among them.

The main contributions of this paper are as follows.

- We propose and analyze quantitatively a novel concurrent-push scheduling algorithm for scheduling disk retrieval and network transmission in parallel video servers.
- We show that a simple extension of the server-push service model for parallel video servers does not perform well, and we propose a new asynchronous grouped sweeping scheme (AGSS) to substantially improve the system performance.
- We propose a new subschedule striping scheme (SSS) to increase the scalability of the architecture even further.

Using numerical results with realistic assumptions, we show that the resultant architecture can be scaled up to more than 10 000 concurrent users.

The rest of this paper is organized as follows. Section II presents the parallel video server architecture and the concurrent-push scheduling algorithm. Section III analyzes the performance of the architecture under the concurrent push algorithm. Section IV presents and analyzes the AGSS. Section V presents the SSS scheme. Section VI evaluates the system performance using numerical results and uses examples to discuss the scalability of the architecture. Section VII reviews some related works and compares them with our approach. Section VIII draws conclusions.

## II. System Architecture

A parallel video server is composed of multiple independent servers connected by an interconnection network (Fig. 1). Each server has a separate CPU, memory, disk storage, and network interface. This *share-nothing* approach ensures that the scalability of the system will not be limited by resource

Manuscript received January 4, 1998; revised August 4, 1998. This paper was recommended by Associate Editor F. Pereira.

The author is with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, N.T., Hong Kong (e-mail: jacklee@computer.org).

Publisher Item Identifier S 1051-8215(99)02959-6.

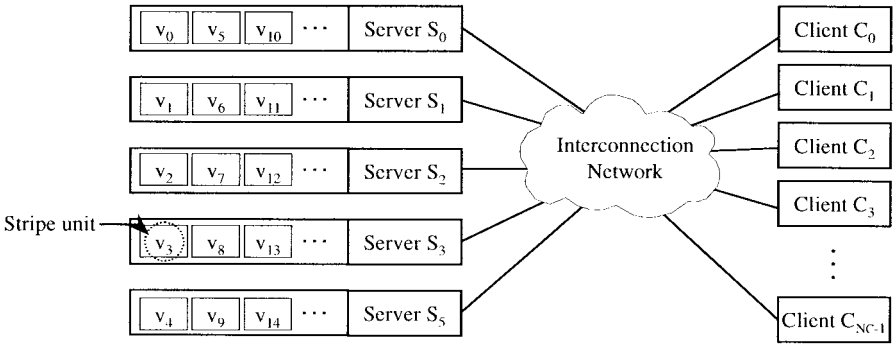



Fig. 1. Architecture of a (five-server) parallel video server.

contention. The interconnection network can be implemented using off-the-shelf packet switches like FastEthernet or asynchronous transfer mode (ATM) switches. We denote the number of servers in the system by $N_S$ and the number of clients by $N_C$. Hence the client-server ratio, denoted by $\Lambda$, is $N_C/N_S$. The following sections present the server striping algorithm, the service model, and the scheduling algorithm employed in the parallel video server studied in this paper.

### A. Server Striping

The principle behind the parallel video server architecture is the striping of a video stream across all servers in the system. A server's storage space is divided into fixed-size stripe units of $Q$ bytes each. Each video title is then striped into blocks of $Q$ bytes and stored into the servers in a round-robin manner, as shown in Fig. 1. This fixed-size block-striping algorithm is called space striping [7], as opposed to striping in units of video frames, called time striping.

Space striping significantly simplifies the process of striping video streams encoded using interframe compression algorithms (e.g., MPEG), where frame size varies considerably for different frame types. Since a stripe unit in space striping is significantly smaller than a video title (kilobytes versus megabytes), this enables fine-grain load sharing (as opposed to course-grain load sharing in data partition) among servers. Moreover, the loads are evenly distributed over all servers independent of the skewness in video retrievals.

### B. Service Model

Service model refers to the way video data are scheduled and delivered to the client. There are two service models in common use: *client pull* and *server push*. In the client-pull model, a client periodically sends a request to a server to retrieve video data. In this model, the data flow is driven by the client. In the server-push model, the server schedules the periodic retrieval and transmission of video data once a video session is started.

The server-push model is common among studies on single-server VoD systems [1]–[4], [8]. This model allows one to design periodic schedulers [8] to optimize disk and network utilization. In the next section, we present an extension of this service model for use in parallel video servers.

### C. Scheduling Algorithm

In this paper, we propose a *concurrent-push* algorithm for scheduling disk retrievals and network transmissions at the servers. The principle behind the concurrent-push algorithm is to let all servers continuously transmit data to a client concurrently. We assume that the average video rate is homogenous for all clients and is denoted by $R_V$. Since there are a total of $N_S$ servers, each server only needs to transmit at a reduced rate of $R_V/N_S$ to maintain an aggregate data rate of $R_V$.

Fig. 2 depicts the scheduling algorithm for disk retrievals and network transmissions at each server in the system. For each video session, one block of $Q$-bytes video data is retrieved into a disk buffer in each disk service round. To reduce seek overhead, requests within a service round can be served using the SCAN or the C-SCAN disk-arm scheduling algorithms [9]. The retrieved video block is then passed to a network buffer for transmission in the next round.[1] Therefore, if the disk service round is shorter than one transmission round, a video block will always be ready for transmission. We analyze the performance of the system under this scheduling algorithm in the next section.

## III. Analysis of the Concurrent-Push Algorithm

In general, the internal clock of each autonomous server in the system is not precisely synchronized. Therefore, the scheduling algorithm must take this server asynchrony into account and compensate accordingly. We define *clock jitter* as the difference between the internal real-time clocks of two servers. Many algorithms for controlling clock jitter between distributed computers have been studied [10], [11] and hence will not be pursued further here. We simply assume that the maximum clock jitter between any two servers in the system is bounded and is denoted by $\tau$. For simplicity, we ignore network delay jitter in this paper. Assuming that network delay jitter is bounded [which is true in ATM networks with quality of service (QoS) guarantees], it is easy to see that the effect of network delay jitter can be incorporated into our performance model in the same way as clock jitter, and the same derivations are still valid.

[1] To avoid data-copying overhead, in practice a disk buffer is just passed to the network subsystem for transmission, while network buffers finishing transmission will be recycled as disk buffers for retrieving new data from the disk.

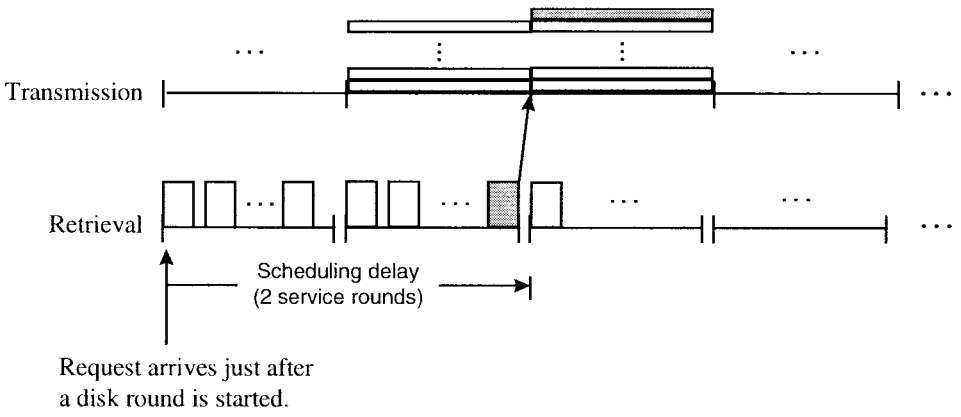



Fig. 2. Scheduling disk retrieval and network transmission at server.

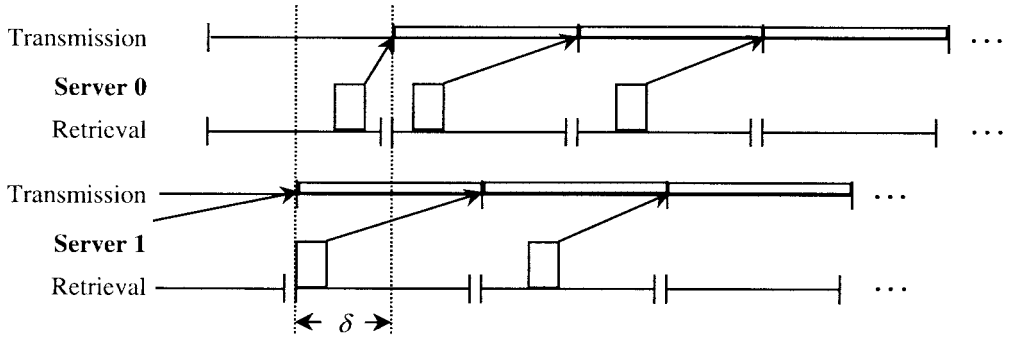



Fig. 3. Service-round misalignment between different servers.

In the following sections, we derive three key performance metrics for evaluating the parallel video server architecture, namely, server buffer requirement, client buffer requirement, and system response time.

### A. Server Scheduling

Under concurrent push, the client will be receiving $N_S$ video blocks simultaneously at an aggregate rate of $R_V$. The average filling time, defined as the time to completely transmit a video block of $Q$ bytes, is given by

$$T_F = \frac{N_S Q}{R_V}. \tag{1}$$

On the other hand, each server will be serving at most $\Lambda N_S$ concurrent video sessions. Under the SCAN disk scheduler, $\Lambda N_S$ video blocks will be retrieved in each service round for transmission at a rate of $R_V/N_S$ per video stream. Hence the duration of a service round is equal to $T_F$ in (1), and two buffers are needed for each video stream for a total of $2\Lambda N_S Q$-bytes buffers at each server.

As server clocks are not synchronous, the service round of the servers may not be aligned (see Fig. 3). Without loss of generality, we assume that a video title is striped with block zero storing at server zero. Let $T_{i,j}$ be the time server $i$ $(0 \le i < N_S)$ starts transmitting the $(jN_S + i)$th block of a video stream. Then we can formally define *transmission jitter* as follows:

$$\delta = \max\{|T_{i,j} - T_{k,j}| \mid \forall i, k, j\}. \tag{2}$$

It may appear that the maximum clock jitter $\tau$ also bounds the transmission jitter. However, it turns out that the transmission jitter depends not only on the clock jitter but also on the arrival time of a new video session request, as depicted in Fig. 4. We derive the upper bound for the transmission jitter in Theorem 1 below.

*Theorem 1:* Assume that new-session requests arrive at all servers at the same time; then the transmission jitter is bounded by

$$\delta \le T_F. \tag{3}$$

*Proof:* Please refer to the Appendix. □

This bound on transmission jitter will be used in Sections III-C and III-D to derive the amount of buffer required at the client to prevent buffer underflow and overflow, respectively.

### B. Video Block Consumption Model

Many studies on VoD systems assume that video data are consumed periodically by the video decoder. However, our experience in programming some off-the-shelf hardware and software video decoders reveals that the decoder consumes fixed-size data blocks only quasi-periodically.

Given the average video data rate $R_V$ and block size $Q$, the average time for a video decoder to consume a single block is

$$T_{\text{avg}} = \frac{Q}{R_V}. \tag{4}$$

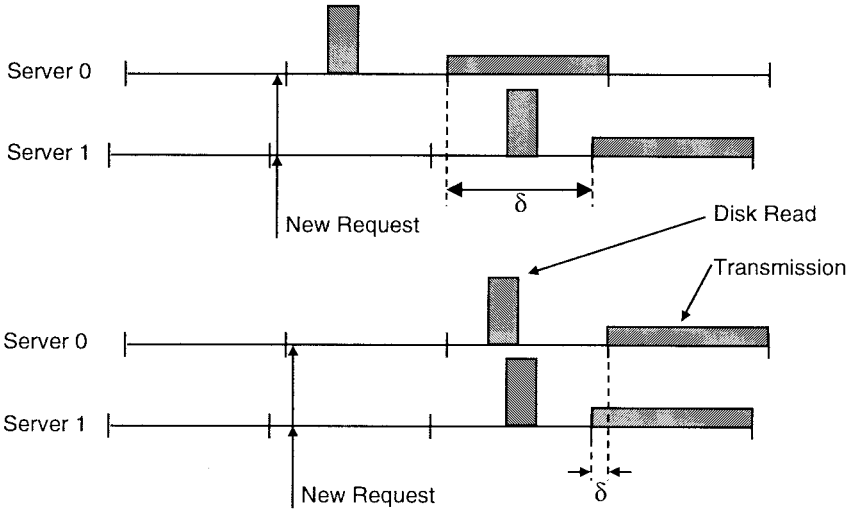



Fig. 4. Transmission jitter depends on both clock jitter and request arrival time.

To quantify the randomness of video block consumption time, we first define a few notations.

*Definition 1:* Let $T_i$ be the time the video decoder starts decoding the $i$th video block; then the decoding-time deviation of video block $i$ is defined as

$$T_{\mathrm{DV}}(i) = T_i = iT_{\mathrm{avg}} - T_0 \tag{5}$$

and decoding is late if $T_{\mathrm{DV}}(i) > 0$ and early if $T_{\mathrm{DV}}(i) < 0$.

The maximum lag in decoding $T_L$ and maximum advance in decoding $T_E$ are defined as

$$T_L = \max\{T_{\mathrm{DV}}(i) \mid \forall i \geq 0\} \tag{6}$$

$$T_E = \min\{T_{\mathrm{DV}}(i) \mid \forall i \geq 0\}. \tag{7}$$

The peak-to-peak decoding-time deviation is defined as

$$T_{\mathrm{DV}} = T_L - T_E. \tag{8}$$

Assume that the bounds $T_L$ and $T_E$ are known. The time between the consumption of two video blocks $i$ and $j$ $(j > i)$ will be bounded by

$$\max\{((j-i)T_{\mathrm{avg}} - T_{\mathrm{DV}}), 0\} \leq t \leq ((j-i)T_{\mathrm{avg}} + T_{\mathrm{DV}}). \tag{9}$$

We use buffers at the client to absorb these variations to prevent buffer underflow and overflow during playback. Let there be $L_C = (Y + Z)$ buffers (each $Q$ bytes) at the client, organized as a circular buffer. The client starts video playback once the first $Y$ buffers are completely filled with video data. We prefill buffers before playback to avoid buffer underflow, and reserve the last $Z$ buffers for incoming data to avoid buffer overflow.

### C. Buffer Needed to Prevent Underflow

Since all $N_S$ servers transmit data to a client concurrently, the client will be receiving $N_S$ video blocks simultaneously. Hence $Y$ must be a multiple of $N_S$. We let $y = Y/N_S$ and consider groups of $N_S$ buffers in the follow derivations (i.e., group zero consists of blocks zero to $N_S - 1$, group one consists of blocks $N_S$ to $2N_S - 1$, and so on).

Among the $N_S$ servers, let the earliest transmission for the first round start at time $t_0$; then the last transmission for the first round must start at time $t_0 + \delta$. Therefore, the time for video block group $i$ to be completely filled, denoted by $F(i)$, is bounded by

$$\begin{aligned}\left((i+1)T_F + t_0 + f^-\right) &\leq F(i) \\ &\leq \left((i+1)T_F + t_0 + \delta + f^+\right)\end{aligned} \tag{10}$$

where $f^+$ $(f^+ \geq 0)$ and $f^-$ $(f^- \leq 0)$ are used to model the maximum transmission-time deviation due to randomness in the system, including transmission rate deviation, CPU scheduling, bus contention, etc.

Since the client starts playing video after filling the first $y$ groups of buffers, the playback time for video block group 0 is simply $F(y-1)$. From Section III-B, setting $T_0 = F(y-1)$, then the playback time for video block group $i$, denoted by $P(i)$, is bounded by

$$\begin{aligned}\{iN_ST_{\mathrm{avg}} + F(y-1) + T_E\} &\leq P(i) \\ &\leq \{iN_ST_{\mathrm{avg}} + F(y-1) + T_L\}.\end{aligned} \tag{11}$$

To guarantee video playback continuity, we must ensure that a video block group arrives before the playback deadline. In the worst case, the latest filling time must be smaller than the earliest playback time, i.e.,

$$\max\{F(i)\} < \min\{P(i)\}. \tag{12}$$

Now, for the left-hand side, noting that $N_ST_{\mathrm{avg}} = T_F$ [see (1) and (4)], we then have

$$\max\{F(i)\} = (i+1)T_F + t_0 + \max\{\delta\} + f^+. \tag{13}$$

Using the upper bound for $\delta$ from Theorem 1, we obtain

$$\begin{aligned}\max\{F(i)\} &= (i+1)T_F + t_0 + T_F + f^+ \\ &= (i+2)T_F + t_0 + f^+.\end{aligned} \tag{14}$$

Similarly, the right-hand side is

$$\begin{aligned}\min\{P(i)\} &= iN_S T_{\text{avg}} + \min\{F(y-1)\} + T_E \\ &= iT_F + yT_F + t_0 + f^- + T_E. \end{aligned} \quad (15)$$

Merging (14) and (15), we then have

$$(i+2)T_F + t_0 + f^+ < (i+y)T_F + t_0 + f^- + T_E. \quad (16)$$

Rearranging, we can then obtain $y$

$$y > 2 + \frac{f^+ - f^- - T_E}{T_F}. \quad (17)$$

Knowing the number of groups required, we can then obtain $Y$ from

$$Y = \left\lceil 2 + \frac{f^+ - f^- - T_E}{T_F} \right\rceil N_S. \quad (18)$$

### D. Buffer Needed to Prevent Overflow

On the other hand, to guarantee that the client buffer will not be overflowed by incoming video data, we need to ensure that the $i$th video block group starts playback before the $(i+l-2)$th video block group is completely received, where $l = L_C/N_S$. This is because the client buffers are organized as a circular buffer, and we must have at least one group of $N_S$ free buffers available for video blocks arriving simultaneously from $N_S$ servers. Therefore, we need to ensure that the earliest filling time for group $(i+l-2)$ must be larger than the latest playback time for group $i$

$$\min\{F(i+l-2)\} > \max\{P(i)\}. \quad (19)$$

Using derivations similar to the previous section, we can obtain the number of buffers needed to prevent buffer overflow as

$$Z = \left\lceil 2 + \frac{f^+ - f^- + T_L}{T_F} \right\rceil N_S. \quad (20)$$

### E. System Response Time

Response time is defined as the time from the user's request for a new video session to the time actual video playback starts. This delay comprises two components: *scheduling delay*, and *prefill delay*. Scheduling delay is the time from a client's sending a new-session request to the time transmission starts at the server. It is easy to see that the worst case scheduling delay is two service rounds (see Fig. 2)

$$D_S = \frac{2N_S Q}{R_V}. \quad (21)$$

Prefill delay is the time from when the server starts transmission to when the first $y$ groups of client buffers are fully filled with data. Using (10), the worst case prefill delay can be obtained from

$$D_P = \max\{F(y-1)\} - t_0 \quad (22)$$

or

$$\begin{aligned} D_P &= yT_F + \max\{\delta\} + f^+ = (y+1)T_F + f^+ \\ &= \left(3 + \left\lceil \frac{f^+ - f^- - T_E}{T_F} \right\rceil\right) T_F + f^+. \end{aligned} \quad (23)$$

## IV. Asynchronous Grouped Sweeping Scheme

The results in the previous section reveal an important characteristic of the concurrent-push algorithm, namely, that the server buffer requirement, client buffer requirement, and response time all increase with the number of servers in the system. Therefore, the scalability of the system will be limited by either the economy of memory buffers or the tolerance of the system response time by the user. In this section, we propose an extension of the grouped sweeping scheme (GSS) [12], called asynchronous group sweeping scheme to substantially reduce server buffer requirement and scheduling delay.

### A. Extending the Grouped Sweeping Scheme

The original GSS algorithm in [12] is designed for scheduling retrieval requests in a magnetic disk. The traditional first-in, first-out scheduling algorithm has poor disk utilization in continuous-media applications [8], [9] because in the worst case, the disk arm may need to seek back and forth between the innermost track and the outermost track, thus wasting a lot of time in seeking. Instead, some researchers use the SCAN scheduling algorithm to reduce seek-time overhead by serving requests while the disk arm scans across the disk surface. However, this approach requires two buffers per stream because requests may be served out of order, and in the worst case, two requests for the same stream may be served in a back-to-back manner.

The GSS algorithm is designed to strike a balance between minimizing seek-time overhead and minimizing buffer requirement by serving streams in groups. Streams within a group are served using SCAN to reduce seek-time overhead, while the groups are served in a fixed order to reduce buffer requirement. By varying the number of groups, one can trade off disk utilization against buffer requirement.

To extend GSS for use in parallel video servers, we propose dividing a service round into $G = gN_S$ groups, where $g$ can be determined using the single-server model as in [12] to minimize buffer requirement while still meeting the playout requirement. Assume that a single server can serve at most $\Lambda$ video sessions; then each group serves up to $(\Lambda/g)$ video sessions. It is easy to see that this holds for two or more servers as well. Therefore, the number of disk buffers needed is reduced from $\Lambda N_S$ to $\Lambda$, though we still need $\Lambda N_S$ network buffers because a video block is transmitted at a lower data rate of $R_V/N_S$. Under this extended GSS algorithm, the total amount of server buffer required will be

$$B_{\text{server}} = QN_S\Lambda\left(1 + \frac{1}{G}\right). \quad (24)$$

### B. Uneven Group Assignment and Admission Scheduling

The AGSS algorithm described in the previous section has a subtle problem when the servers in the system are not clock synchronized. Fig. 5 illustrates the problem using the arrivals of two new-session requests. As shown in the figure, while server zero assigns the two new sessions to different groups, server one assigns them to the same group. This can

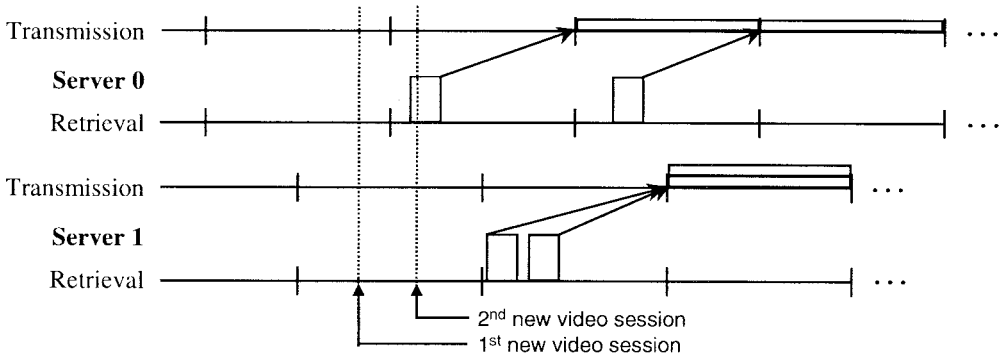



Fig. 5. Uneven service-round assignments.

occur because each server assigns the new session to a group according to its own internal clock, which may be different from other servers due to clock jitter. Eventually, the group occupancy among servers may deviate in such a way that one server can accept a new video session immediately while others have to wait for an available group, thereby increasing the transmission jitter.

To reduce the transmission jitter (which also reduces buffer requirement at the client), we propose adding an admission scheduler to handle group assignment for new-session requests. To initiate a new video session, a client will first send a request to the admission scheduler, which maintains the same clock-jitter bound with the servers. As new sessions are assigned solely according to the admission scheduler's clock, the scenario depicted in Fig. 5 will not occur. To ensure that the assigned group has not started in any of the servers due to clock jitter, the admission scheduler adds an extra delay to the assignment, stated in the following theorem.

*Theorem 2:* If the admission scheduler delays the start of a new video session by

$$\Omega = \left\lceil \frac{\tau G}{T_F} \right\rceil + 1 \tag{25}$$

groups, then it guarantees that the assigned group has not started in any of the $N_S$ servers.

*Proof:* Please refer to the Appendix. □

Note that if the assigned group is full, the admission scheduler will sequentially check the subsequent groups until an available group is found.

### C. Client Buffer Requirement

As the admission scheduler already guarantees that a new video session will be assigned to the same group in all servers, the scenario in Fig. 5 could not occur, and the transmission jitter will be the same as the clock jitter. Hence the client buffer requirement derived in Section III becomes

$$Y = \left\lceil 1 + \frac{\tau + f^+ - f^- - T_E}{T_F} \right\rceil N_S \tag{26}$$

$$Z = \left\lceil 1 + \frac{\tau + f^+ - f^- + T_L}{T_F} \right\rceil N_S. \tag{27}$$

### D. System Response Time

The scheduling delay under the AGSS algorithm depends on the occupancy of the AGSS groups. Specifically, if a group as calculated from Theorem 2 is fully occupied, the new video session must be delayed until the next available group. In the worst case, the transmission of the first video block is delayed for $(N_S + \Omega)$ groups

$$D_S = \left( N_S + \left\lceil \frac{\tau G}{T_F} \right\rceil + 1 \right) . \frac{Q}{R_V}. \tag{28}$$

To better evaluate the scheduling delay, we derive the average scheduling delay under a given system load. Assume that video sessions start independently and with equal likelihood at any time. Then a video session can be assigned to any one of the $G$ groups with equal probability. Let there be $n$ active video sessions and $G$ groups; then the number of ways to distribute these $n$ video sessions among $G$ groups is a variant of the urn-occupancy distribution problem and is given by [13] as

$$N(n, G, \Lambda) = \sum_{j=0}^{G} (-1)^j \binom{G}{j} \binom{G + n - j(\Lambda + 1) - 1}{G - 1}. \tag{29}$$

To obtain the probability of having $m$ fully occupied groups, we first notice that there are $\binom{G}{m}$ possible combinations of picking $m$ fully occupied groups among $G$ groups. Given that there are $n$ active video sessions and $m$ fully occupied groups, The number of ways to distribute the remaining $(n - m\Lambda/g)$ video sessions among the remaining $(G-m)$ groups with none of those groups fully occupied can be obtained from (29) as $N(n - m\Lambda/g, G - m, (\Lambda/g) - 1)$. Hence the total number of ways for exactly $m$ of the groups to be fully occupied is given by

$$N_{\text{full}}(n, m) = \binom{G}{m} N\left( n - m\frac{\Lambda}{g}, G - m, \frac{\Lambda}{g} - 1 \right). \tag{30}$$

The probability of having $m$ fully occupied groups given $n$ active video sessions can then be obtained from

$$P_{\text{full}}(n, m) = \frac{N_{\text{full}}(n, m)}{N(n, G, \Lambda)}. \tag{31}$$

Knowing this, we can derive the average scheduling delay in the following way. Given that $m$ out of $G$ groups are

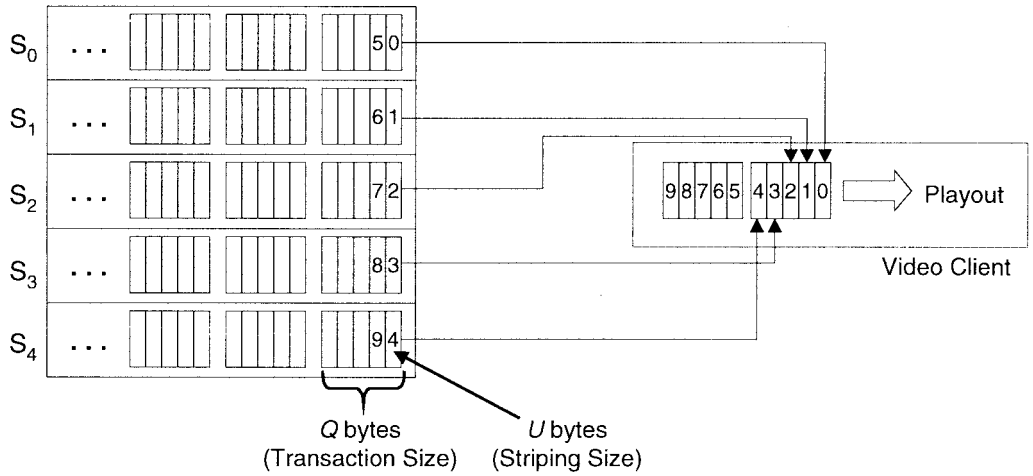



Fig. 6. Data organization in subschedule striping.

fully occupied, the probability of the assigned group's being available (not fully occupied) is given by

$$V_0 = \frac{G-m}{G}. \tag{32}$$

Hence $P_0 = (1 - V_0)$ will be the probability of the assigned group's being fully occupied. It can be shown that the probability of a client's waiting $k$ additional groups provided that the first $k$ assigned groups are all fully occupied is

$$V_k = \Pr\{(k+1)\text{th group available} \mid P_k\} = \frac{G-m}{G-k}, \quad 1 \le k \le m \tag{33}$$

and the probability that the first $k$ groups are all fully occupied is

$$P_k = \prod_{i=0}^{k-1}\left(\frac{m-k}{G-i}\right) = \frac{m!(G-k)!}{G!(m-k)!}, \quad 1 \le k \le m. \tag{34}$$

Hence we can solve for the probability of a client's having to wait $k$ additional groups, denoted by $W_k$, from

$$\begin{aligned} W_k &= \Pr\{(k+1)\text{th group free} \mid P_k\}P_k \\ &= \frac{(G-m)m!(G-k-1)!}{G!(m-k)!}, \quad 1 \le k \le m. \end{aligned} \tag{35}$$

Therefore, given the number of groups that are fully occupied $m$, the average number of groups a client has to wait can be obtained from

$$W_{\text{avg}}(m) = \sum_{k-1}^{m} kW_k + \left\lceil \frac{\tau G}{T_F} \right\rceil + 1. \tag{36}$$

Similarly, given the number of active video sessions $n$, the average number of groups a client has to wait can be obtained from (31) and (36) as follows:

$$M_{\text{avg}}(n) = \sum_{j=1}^{G-1} W_{\text{avg}}(j)P_{\text{full}}(n,j) \tag{37}$$

and the corresponding average scheduling delay given a system utilization of $n$ is

$$D_S = \frac{M_{\text{avg}}(n)Q}{R_V}. \tag{38}$$

As the admission scheduler reduces the transmission jitter to equal the clock jitter, the new prefill delay can be obtained by replacing $\delta$ with $\tau$ in (23)

$$D_P = \left(2 + \left\lceil \frac{\tau + f^+ - f^- - T_E}{T_F} \right\rceil\right)T_F + f^+. \tag{39}$$

## V. Subschedule Striping Scheme

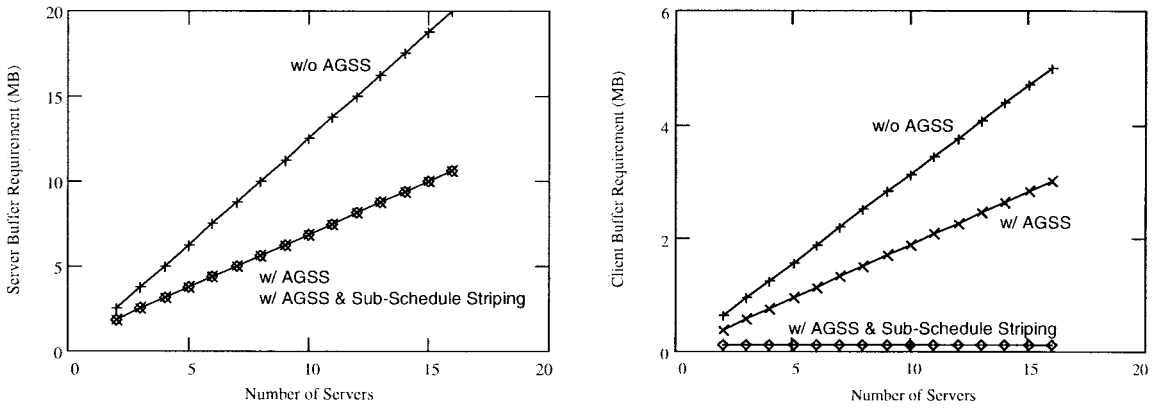
The AGSS algorithm presented in the previous section substantially reduces the server buffer requirement as well as the scheduling delay. However, the client buffer requirement, and consequently the prefill delay, are only slightly reduced as a side effect of the admission scheduler. In this section, we consider another modification to the concurrent-push algorithm that can substantially reduce the client buffer requirement and the prefill delay.

Specifically, the analysis in Section III reveals that the main reason for the increase in client buffer requirement with the number of servers stems from the increase in the average filling time in (1). This suggests that we can reduce the buffer requirement by using smaller striping size $Q$. However, as the server retrieves data from the disk in units of $Q$ bytes, reducing the striping size will adversely affect disk-retrieval efficiency.

To solve this problem, we propose decoupling the transaction size for disk retrieval and transmission from the striping size—*subschedule striping*. In particular, we maintain the disk transaction size at $Q$ bytes but use a striping size (denoted by $U$) inversely proportional to the number of servers in the system (Fig. 6)

$$U = Q/N_S. \tag{40}$$

Hence the disk will retrieve $N_S$ stripe units in a single transaction. Note that the client continues to consume video data in blocks of $Q$ bytes, and hence the video-block consump-




(a) (b)

Fig. 7. (a) Server buffer requirement versus number of servers. (b) Client buffer requirement versus number of servers.

tion model in Section III-B remains valid. However, a video block now contains stripe units transmitted from all $N_S$ servers (Fig. 6) rather than from a single server as in the original algorithm. Consequently, the client buffer sizes $Y$ and $Z$ no longer need to be multiples of $N_S$.

SSS requires no modification to the server as the transaction size remains the same. Therefore, the server buffer requirement as well as the scheduling delay are the same as before.

To model the effect on the client buffer requirement, we note that a $Q$-bytes video block comprises fragments from all $N_S$ servers. Hence the filling time for a video block would be affected by the transmission jitter among servers. Specifically, the filling time for video block $i$ of a video stream started at time $t_0$ is bounded by

$$\begin{aligned}\left((i+1)T_{\text{avg}} + t_0 + f^-\right) &\le f(i) \\ &\le \left((i+1)T_{\text{avg}} + t_0 + f^+ + \tau\right).\end{aligned} \tag{41}$$

Using similar derivations, the client buffers needed to prevent underflow and overflow can be found as

$$Y > 1 + \left(\frac{f^+ - f^- - T_E + \tau}{T_{\text{avg}}}\right) \tag{42}$$

$$Z > 1 + \left(\frac{f^+ - f^- + T_L + \tau}{T_{\text{avg}}}\right) \tag{43}$$

and the time to prefill the first $Y$ client buffers is

$$D_P = YT_{\text{avg}} + f^+ + \tau. \tag{44}$$

Now both the client buffer requirement and prefill delay no longer depend on the number of servers in the system.

## VI. Performance Evaluation

In this section, we evaluate the performance of the parallel video server architecture studied in this paper using numerical results. Table I lists the values for the key system parameters used in the calculation. The parameters $T_E$ and $T_L$ are determined empirically by collecting the video block consumption times of a hardware MPEG-1 decoder.

TABLE I
System Parameters Used in Performance Evaluation

| System Parameter | Symbol | Value |
|---|---|---|
| Video block size | $Q$ | 65536 Bytes |
| Video data rate | $R_V$ | 150KB/s |
| Maximum advance in decoding time | $T_E$ | -130ms |
| Maximum lag in decoding time | $T_L$ | 160ms |
| Client-Server ratio | $\Lambda$ | 10 |
| Transmission time deviation | $f^-, f^+$ | 0ms |
| Server clock jitter | $\tau$ | 100ms |
| AGSS parameter | $g$ | 1 |

### A. Server Buffer Requirement

Fig. 7(a) plots the per-server buffer requirement versus the number of servers in the system. We can observe that AGSS substantially reduces the buffer requirement. SSS has no effect on the server buffer requirement. Despite the reduction achieved by AGSS, the server buffer requirement still increases with the number of servers. This poses one limitation on the ultimate scalability of the system (to be discussed in Section VI-D). Depending on the relative cost of memory and disk bandwidth, one may reduce system cost by trading disk efficiency for smaller server buffer requirement.

### B. Client Buffer Requirement

Fig. 7(b) plots the client buffer requirement versus the number of servers in the system. The figure shows that AGSS substantially reduces the client buffer requirement, but it still increases linearly with the number of servers. With the addition of subschedule striping, the client buffer requirement is constant regardless of the number of servers in the system. This is a crucial property, as it would be impractical to upgrade all clients whenever more servers are added to the system in practice.

From (42) and (43), it is easy to see that the client buffer requirement is insensitive to the server clock jitter. As an example, for a 16-servers system with AGSS and SSS, the client buffer requirement is only 384 KB for a server clock jitter as large as 1000 ms.

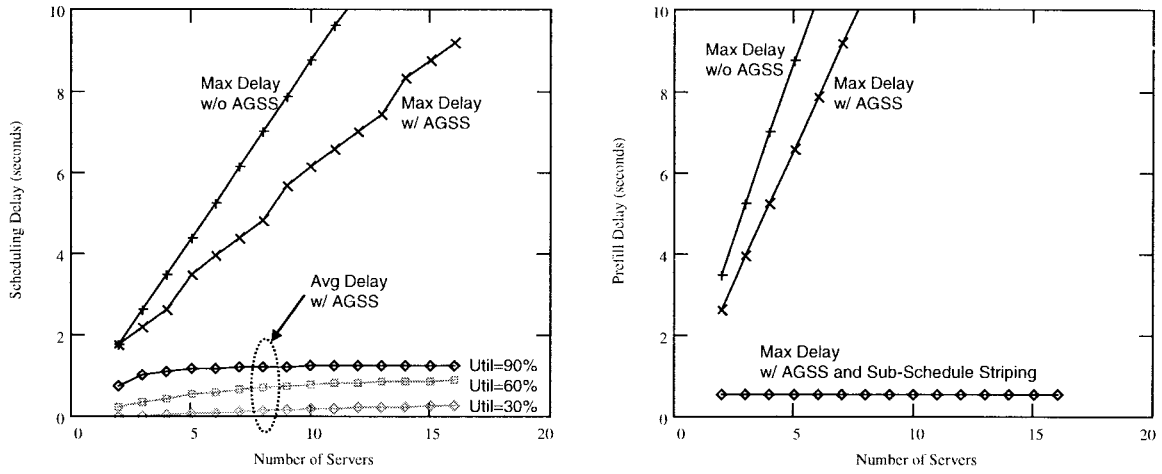



Fig. 8. (a) Scheduling delay versus number of servers. (b) Prefill delay versus number of servers.

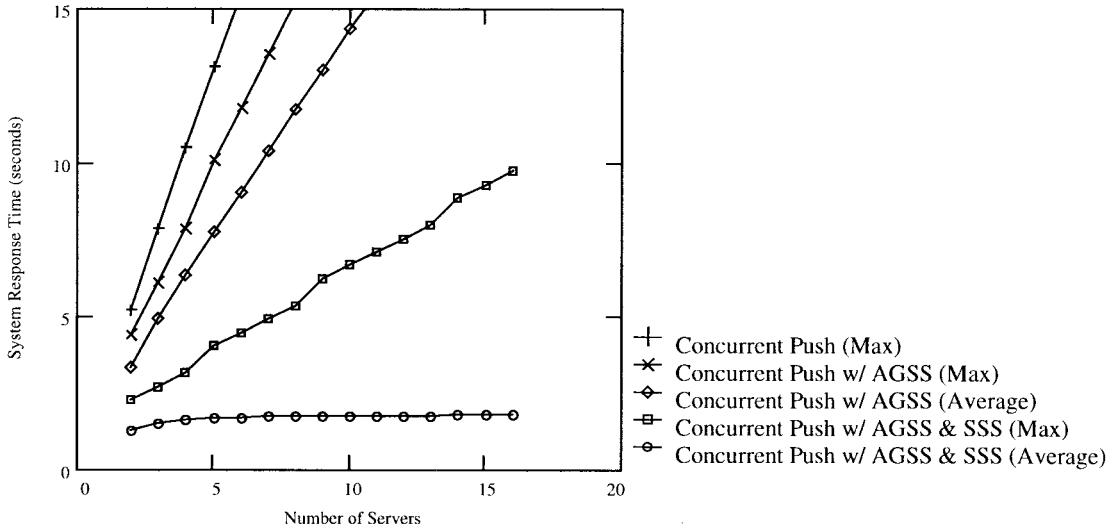



Fig. 9. System response time versus number of servers.

### C. System Response Time

We first plot scheduling delay versus the number of servers in the system in Fig. 8(a). The case for SSS is not plotted, as it has no effect on the scheduling delay. Note that the worst case scheduling delay is substantially reduced by AGSS, especially for a large number of servers. Moreover, the average scheduling delay with AGSS is even smaller and stays relatively constant regardless of the number of servers in the system. For example, with AGSS striping in a 16-servers system, the average delay is only 1.26 s for system utilization as high as 90%, even though the worst case is 9.18 s. The worst case delay is even larger (13.98 s) without AGSS. Hence, with AGSS, we can maintain a reasonably low scheduling delay by operating the system to within, say, 90% of the total capacity.

Fig. 8(b) plots the prefill delay versus the number of servers in the system. The results show that the prefill delay is also reduced by AGSS because the worst case transmission jitter $T_F$ is larger than the clock jitter $\tau$. More important, by using subschedule striping, the prefill delay becomes completely independent of the number of servers in the system.

Last, we plot the total system response time in Fig. 9. Clearly, the proposed AGSS and SSS can effectively maintain a small system response time (1.8 s for $N_S = 16$ at 90% utilization) even if the number of servers is large.

### D. Scalability

The results in the previous sections have shown that both the client buffer requirement and the system response time can be maintained low irrespective of the number of servers in the system. The server buffer requirement is the only factor that increases with more servers. This factor will certainly limit the ultimate scalability of the system. Nowadays, it is common to install 256 MB or more memory in a PC-based server, as the price of memory has dropped substantially. Under our system parameters and ignoring operating system overhead, a 256-MB memory size will limit the scalability of the parallel video server architecture to a maximum of 408 servers serving

a total of 3672 concurrent video sessions at 90% utilization. If 1-GB memory is available, the architecture can be scaled up to 14 400 concurrent video sessions using a client–server ratio of 250 at 90% utilization.

A second, more subtle limiting factor is due to the SSS scheme. Under this scheme, the client must resequence the incoming data by copying $U$-bytes stripe units into the client buffer (Fig. 6). Hence the processing overhead will likely increase with smaller striping size. Our previous experiences [14] showed that processing overhead remains practical for software implementations using i486-based PC's for striping size as small as 1 KB. This limits $N_S$ to 64. For larger systems, we can use more powerful server hardware with a larger client–server ratio to avoid hitting this limit. In the previous example with 1-GB memory, we increase the client–server ratio to 250 so that the total number of servers required stays within 64 ($(14\,400/0.9)/250 = 64$) to avoid excessive processing at the client. Clearly, we will be able to use smaller striping size to extend this limit as CPU speed improves.

## VII. Related Works

There has been an increasing interest in designing parallel video servers. Related studies have been conducted by Bernhardt and Biersack [15], *et al.* [16], Buddhikot and Parulkar [17], Lee and Wong [18], [19], Reddy [20], Tewari *et al.* [21], and Wu and Shu [22]. A general introduction to parallel video servers can be found in [7]. We summarize below the major differences between the previous works and the approach proposed in this paper.

The studies by Reddy [20], Tewari *et al.* [21], and Wu and Shu [22] are based on architectures where one or more intermediate delivery nodes are used to merge video data from multiple servers for delivery to clients. Conversely, in our architecture, servers transmit video data to a client without passing through any intermediate node. Our approach eliminates the extra hardware needed to run the intermediate delivery nodes.

Lee and Wong [18], [19] have designed and implemented a parallel video server for local-area networks. Their system employs the client-pull service model rather than the server-push service model studied in this paper.

The study by Buddhikot and Parulkar [17] also employs the server-push service model. However, the servers in their system transmit bursts of data in a staggered manner rather than continuously at a constant rate as our architecture does. Therefore, their system cannot take advantage of the QoS guarantee provided by current ATM hardware, and must rely on a proprietary ATM switch to precisely synchronize the server clocks and merge the data bursts into a single continuous data stream for delivery to clients. In designing our architecture, we have deliberately avoided the use of proprietary hardware to lower the cost of the system. By sending data continuously, our architecture can take advantage of QoS guarantees provided by current ATM hardware. Moreover, our architecture is robust to server clock jitter and hence allows conventional distributed clock synchronization algorithms to be used for server clock synchronization.

Bernhardt and Biersack [15] and Biersack *et al.* [16] have also studied the problem for server-push designs and proposed algorithms to compensate for network delay differences and clock drifts. Unlike our architecture, their system stripes video data in units of frames rather than in fixed-size blocks. They did not consider variations in video block consumption times and assumed constant consumption time in [15]. Their study also did not consider scheduling issues at the servers. Our study has revealed the limitations of a straightforward extension of the server-push model, and we proposed the AGSS with admission scheduling and subschedule striping schemes to extend the scalability of the architecture. Last, our study has quantitatively analyzed the system response time, which is a key performance metric in practice.

## VIII. Conclusion

In this paper, we have proposed and analyzed a parallel video server architecture for designing scalable video-on-demand systems. The proposed architecture employs fixed-size block striping and the server-push service model. To schedule disk retrievals and transmissions, we proposed a concurrent-push scheduling algorithm where video data are continuously transmitted from all servers to a client station. This constant-bit-rate traffic produced by the algorithm enables us to take advantage of the QoS guarantees provided by today's ATM networks. To extend the scalability of the architecture, we introduced the asynchronous grouped sweeping scheme and the subschedule striping scheme into the architecture. Results showed that the resultant architecture can be scaled up to more than 10 000 concurrent users with acceptable buffer requirement and system response time.

Building video-on-demand systems upon parallel server architecture not only breaks through the capacity limit of a single server but also opens the way to fault-tolerant system designs. An early study [19] has already demonstrated the feasibility of achieving server-level fault tolerance by introducing data redundancy among the servers. We are currently investigating ways to integrate fault tolerance into the architecture studied in this paper.

## Appendix

*Proof of Theorem 1:* Let server zero start the first service round at time $t_0$. Since the server clocks are not exactly synchronized, we let $d_i$ be the clock difference between server $i$ and server zero. Hence $d_0 = 0$ and $\max\{|d_i - d_j| \mid \forall i, j\} = \tau$, and server $i$ will start service round $j$ at time $(t_0 + d_i + jT_F)$. Let $t_{\text{new}}$ be the time a new-session request arrives at the servers. Then the request will arrive at server $i$ during round $v_i$

$$v_i = \left\lfloor \frac{t_{\text{new}} - (t_0 + d_i)}{T_F} \right\rfloor \tag{45}$$

and the first video block will be retrieved at round $(v_i + 1)$ and transmitted at round $(v_i + 2)$. Hence the transmission jitter

between server $i$ and server $k$ for stripe $j$ can be expressed as

$$\delta_{i,k,j} = (t_0 + d_i + (v_i + 2 + j)T_F) - (t_0 + d_k + (v_k + 2 + j)T_F). \quad (46)$$

Substituting (45) into (46), we have

$$\delta_{i,k,j} = \left(d_i + \left\lfloor \frac{t_{\text{new}} - (t_0 + d_i)}{T_F} \right\rfloor T_F\right) - \left(d_k + \left\lfloor \frac{t_{\text{new}} - (t_0 + d_k)}{T_F} \right\rfloor T_F\right). \quad (47)$$

Without loss of generality, we can assume $d_i \geq d_k$ and let $H = t_{\text{new}} - (t_0 + d_i)/T_F$. Then we have

$$\begin{aligned}\delta_{i,k,j} &= (d_i + \lfloor H \rfloor T_F) - \left(d_k + \left\lfloor H + \frac{(d_i - d_k)}{T_F} \right\rfloor T_F\right) \\ &= (d_i - d_k) + T_F\left(\lfloor H \rfloor - \left\lfloor H + \frac{(d_i - d_k)}{T_F} \right\rfloor\right). \quad (48)\end{aligned}$$

Noting that $\lfloor x + y \rfloor \geq \lfloor x \rfloor + \lfloor y \rfloor$, we have

$$\begin{aligned}\delta_{i,k,j} &\leq (d_i - d_k) + T_F\left(\lfloor H \rfloor - \lfloor H \rfloor - \left\lfloor \frac{(d_i - d_k)}{T_F} \right\rfloor\right) \\ &= (d_i - d_k) - T_F \left\lfloor \frac{(d_i - d_k)}{T_F} \right\rfloor. \quad (49)\end{aligned}$$

Last, making use of the result that $\lfloor x/y \rfloor y \geq (x - y)$, we can then obtain

$$\begin{aligned}\delta_{i,k,j} &= (d_i - d_k) + T_F \left\lfloor \frac{(d_i - d_k)}{T_F} \right\rfloor \\ &\leq (d_i - d_k) - ((d_i - d_k) - T_F) \\ &= T_F \quad (50)\end{aligned}$$

and the result follows. □

*Proof of Theorem 2:* Let the new-session request arrive at the admission scheduler at time $t$ during group $v_{\text{new}} = \lfloor tG/T_F \rfloor$. Then due to clock jitter, the current group at other servers can range from $\lfloor (t - \tau)G/T_F \rfloor$ to $\lfloor (t + \tau)G/T_F \rfloor$. To guarantee that the assigned group has not started in any of the servers implies assigning a group larger than the largest current group in any of the servers, i.e., $s_{\text{new}} = \lfloor (t + \tau)G/T_F \rfloor + 1$. Applying the inequality $\lfloor x + y \rfloor \leq \lfloor x \rfloor + \lceil y \rceil$, we have

$$s_{\text{new}} \leq \lfloor tG/T_F \rfloor + \lceil \tau G/T_F \rceil + 1. \quad (51)$$

Substituting into $v_{\text{new}}$, we have $s_{\text{new}} \leq v_{\text{new}} + \lceil \tau G/T_F \rceil + 1$, and the result follows. □

## ACKNOWLEDGMENT

The author would like to thank the reviewers for their constructive comments in improving this paper for its final form.

## REFERENCES

[1] T. C. Chiueh, C. Venkatramani, and M. Vernick, "Design and implementation of the stony brook video server," Computer Science Department, State University of New York at Stony Brook, Tech. Rep. TR-16, Aug. 1995.
[2] F. A. Tobagi and J. Pang, "StarWorks—A video applications server," in *Proc. IEEE COMPCON Spring'93*, 1993, pp. 4–11.
[3] R. Buck, "The oracle media server for nCube massively parallel systems," in *Proc. 8th Int. Parallel Processing Symp.*, 1994, pp. 670–673.
[4] H. Taylor, D. Chin, and S. Knight, "The magic video-on-demand server and real-time simulation system," *IEEE Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 40–51, 1995.
[5] C. Griwodz, M. Bar, and L. C. Wolf, "Long-term movie popularity models in video-on-demand systems or the life of an on-demand movie," in *Proc. Multimedia'97*, pp. 349–357.
[6] T. D. C. Little and D. Venkatesh, "Popularity-based assignment of movies to storage devices in a video-on-demand system," *ACM Multimedia Syst.*, vol. 2, no. 6, pp. 280–287, 1995.
[7] Y. B. Lee, "Parallel video servers—A tutorial," *IEEE Multimedia Mag.*, vol. 5, no. 2, pp. 20–28, 1998.
[8] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe, "Multimedia storage servers: A tutorial," *IEEE Comput. Mag.*, vol. 28, pp. 40–49, May 1995.
[9] A. L. N. Reddy and J. C. Wyllie, "I/O issues in a multimedia system," *IEEE Comput. Mag.*, vol. 27, pp. 69–74, Mar. 1994.
[10] D. Mills, "Internet time synchronization: The network time protocol," *IEEE Trans. Commun.*, vol. 39, pp. 1482–1493, Oct. 1991.
[11] Z. Yang and T. A. Marsland, Eds., *Global States and Time in Distributed Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1994.
[12] P. S. Yu, M. S. Chen, and D. D. Kandlur, "Grouped sweeping scheduling for DASD-based multimedia storage management," *ACM Multimedia Syst.*, vol. 1, pp. 99–109, 1993.
[13] J. N. Lloyd and K. S. Samuel, *Urn Models and Their Application*. New York: Wiley, 1997, pp. 125–126.
[14] Y. B. Lee and P. C. Wong, "VIOLA—Video on local area networks," in *Proc. 2nd ISMM/IASTED Int. Conf. Multimedia Systems and Applications*, Stanford University, Stanford, CA, Aug. 1995, pp. 101–104.
[15] C. Bernhardt and E. Biersack, "The server array: A scalable video server architecture," *High-Speed Networks for Multimedia Applications*. Norwell, MA: Kluwer, 1996.
[16] E. Biersack, W. Geyer, and C. Bernhardt, "Intra- and inter-stream synchronization for stored multimedia streams," in *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, Hiroshima, Japan, June 17–23, 1996, pp. 372–381.
[17] M. M. Buddhikot and G. M. Parulkar, "Efficient data layout, scheduling and playout control in MARS," in *Proc. NOSSDAV'95*, 1995, pp. 318–329.
[18] Y. B. Lee and P. C. Wong, "A server array approach for video-on-demand service on local area networks," in *Proc. IEEE INFOCOM'96*, San Francisco, CA, Mar. 1996, pp. 27–34.
[19] P. C. Wong and Y. B. Lee, "Redundant array of inexpensive servers (RAIS) for on-demand multimedia services," in *Proc. ICC'97*, Montreal, Canada, June 8–12, 1997, pp. 787–792.
[20] A. Reddy, "Scheduling and data distribution in a multiprocessor video server," in *Proc. 2nd IEEE Int. Conf. Multimedia Computing and Systems*, 1995, pp. 256–263.
[21] R. Tewari, R. Mukherjee, and D. M. Dias, "Real-time issues for clustered multimedia servers," IBM Research Rep. RC20020, June 1995.
[22] M. Wu and W. Shu, "Scheduling for large-scale parallel video servers," in *Proc. 6th Symp. Frontiers of Massively Parallel Computation*, Oct. 1996, pp. 126–133.

**Jack Y. B. Lee** received the B.Eng. and Ph.D. degrees from the Department of Information Engineering, Chinese University of Hong Kong (CUHK), in 1993 and 1997, respectively.

He is an Assistant Professor at the Hong Kong University of Science and Technology (HKUST), Hong Kong. Before joining HKUST in 1998, he was a Visiting Assistant Professor at CUHK for one year. His research interests include distributed multimedia systems, fault-tolerant systems, and Internet computing.