Supporting Server-Level Fault Tolerance in Concurrent-Push-Based Parallel Video Servers

Jack Y. B. Lee

***Abstract*—Parallel video servers have been proposed for building large-scale video-on-demand (VoD) systems from multiple low-cost servers. However, when adding more servers to scale up the capacity, system-level reliability will decrease as failure of any one of the servers will cripple the entire system. To tackle this reliability problem, this paper proposes and analyzes architectures to support server-level fault tolerance in parallel video servers. Based on the concurrent push architecture proposed earlier, this paper tackles three problems pertaining to fault tolerance, namely redundancy management, redundant data transmission protocol, and real-time fault masking. First, redundant data based on erasure codes are introduced to video data stored in the servers, which are then delivered to the clients to support fault tolerant. Despite the success of distributed redundancy striping schemes such as RAID-5 in disk array implementations, we discover that similar schemes extended to the server context do not scale well. Instead, we propose a redundant server scheme that is both scalable, and with lower total server buffer requirement. Second, two protocols are proposed to manage the transmission of redundant data to the clients, namely forward erasure correction which always transmits redundant data, and on-demand correction which transmits redundant data only after a server failure is detected. Third, to enable ongoing video sessions to maintain nonstop video playback during failure, we propose using fault masking at the client to recompute lost video data in real-time. In particular, we derive the amount of client buffer required so that nonstop, continuous video playback can be maintained despite server failures.**

***Index Terms*—Concurrent push, fault tolerance, parallel video server, scheduling algorithm, server failure, server push, video-on-demand.**

I. Introduction

WHILE video-on-demand (VoD) systems have been available for many years, large-scale deployments of VoD services are still uncommon. One reason is the high cost involved in setting up a broadband network infrastructure, acquiring high-capacity video servers, and installing a large number of set-top boxes. For video servers, most large-scale systems available today are of proprietary nature, employing massively parallel processing (MPP) platforms (e.g., nCube [1] and Magic [2]). While these platforms can provide capacities far exceed that of conventional server platforms (e.g., PC server), their costs are inherently higher due to the lower production volume.

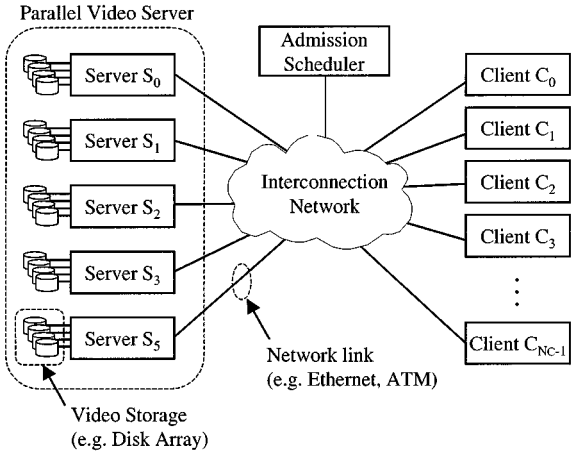

Fig. 1. Architecture of a (5-server) parallel video server.

Recently, parallel video server has been proposed for building large-scale VoD systems from multiple low-cost servers [3]–[17] such as PC-based servers. One such architecture—concurrent push [10]—comprises multiple autonomous servers together with an admission scheduler connected to the clients by an interconnection network (Fig. 1). By dividing a video title into small, fixed-size units, and distributing them to all servers, i.e., *server striping*, this architecture can achieve perfect load balancing among servers and yet does not require video data replication. The servers simultaneously transmit video data to a client continuously at a proportionally reduced rate so that existing ATM quality-of-service (QoS) controls can be employed. Results [10] showed that concurrent push can potentially be scaled up to more than 10 000 concurrent video streams using current PC platforms.

One potential problem with the concurrent push architecture, and any parallel architecture including MPP, is reliability. As the system distributes video data over multiple servers, failure of a single server will cripple the entire system. Worse still, as the system is scaled up to more users, more servers will be needed and consequently the system-wide reliability will decrease accordingly. Drawing similar principles from disk array researches, we propose improving system reliability by means of introducing redundancy into the system.

The major contributions of this study are in tackling three key problems pertaining to supporting fault tolerance under the concurrent push architecture, namely redundancy management, redundant data transmission protocol, and real-time fault

Manuscript received April 12, 1999; revised April 26, 2000. This research was supported in part by the HKSAR Research Grant Council under Grant CUHK6095/99E and by the Area-of-Excellence in IT, a Research Grant from the HKSAR University Grants Council.

The author is with the Department of Information Engineering at the Chinese University of Hong Kong, Shatin, N.T., Hong Kong (e-mail: jacklee@computer.org).

Publisher Item Identifier S 1051-8215(01)00668-1.

masking. First, redundant data based on erasure codes are introduced to video data stored in the servers, which are then delivered to the clients to support fault tolerant. Despite the success of distributed redundancy striping schemes such as RAID-5 [18] in disk array implementations, we discover that similar schemes extended to the server context do not scale well. Instead, we propose a redundant server scheme that is both scalable, and with lower total server buffer requirement. Second, two protocols are proposed to manage the transmission of redundant data to the clients, namely forward erasure correction (FEC), which transmits redundant data even when there is no failure, and on-demand correction (ODC), which transmits redundant data only after a server failure is detected. These two protocols achieve different tradeoffs between bandwidth overhead, implementation complexity, and client buffer requirement. Third, to enable ongoing video sessions to maintain nonstop video playback during failure, we propose using fault masking at the client to recompute lost video data in real-time. In particular, we derive the amount of client buffers required so that nonstop, continuous video playback can be maintained.

The rest of the paper is organized as follows. Section II reviews related works in this area and compares them with this paper. Section III presents an overview of the concurrent push architecture. Section IV presents how redundancy can be introduced into concurrent push by extending the RAID-5 scheme to the server level. Section V presents the FEC protocol for transmitting redundant data. Section VI presents the ODC protocol for transmitting redundant data. Section VII analyzes the FEC protocol by deriving the amount of client buffer required to sustain nonstop video playback despite failure. Section VIII analyzes the ODC protocol by deriving the amount of client buffer required to sustain nonstop video playback despite failure. Section IX presents the redundant server scheme (RSS) that solves the scalability problem in ODC. Section X presents numerical results computed from the derivations and analyzes the sensitivity of the proposed algorithms and protocols to several key system parameters. Finally, Section XI concludes the paper.

II. Related Works

While many studies have investigated disk-level fault tolerance in video servers (e.g., [19]–[22]), only three studies [5], [15], [16] known to the author have investigated server-level fault tolerance in parallel video servers. In this section, we briefly review these studies and compare them with the approach proposed in this paper.

Bolosky *et al.* [5] proposed the use of data mirroring to improve reliability in their Tiger video server (now known as Microsoft NetShow Theatre). Similar to disk mirroring, they proposed storing two copies of every stripe unit at the server nodes. Hence, in case a server fails, rendering one of the copies unavailable, the system can still use the remaining copy for delivery. Clearly, the two copies must reside at two different server nodes so that a node failure will not render both copies unavailable simultaneously. Additionally, Bolosky *et al.*proposed the use of declustering for stripe-unit placement. Briefly speaking, given the number of server nodes, a declustering scheme determines the placement of the replicated stripe units so that additional load in retrieving the backup copies are evenly distributed across all server nodes. Declustering has been studied extensively in the context of disk arrays and the interested readers are referred to [23] for a study of declustering in a disk array.

One tradeoff in data mirroring is doubled storage requirement, which could be expensive for applications like video library or paid-movie service. A subtler tradeoff is the need for declustering. As no known algorithm can automatically produce a declustering scheme for an arbitrary number of servers, this mirroring approach would require more complex capacity planning and data reshuffling when being scaled up.

In comparison, the architecture proposed in this paper does not need full data replication (unless there are only two servers) or declustering. For example, with a server mean time to failure (MTTF) of 50 000 hours and a targeted system MTTF of 10 000 hours (see Section X-D), the redundancy overhead is only around 20%. The striping and placement policy is simple and can be scaled to any number of servers. In addition, although mirroring can sustain single-server failure, the proposed architecture can sustain K servers failing simultaneously. Last but not least, reliability of the Tiger video server will inevitably decrease as the system is scaled up because only one server failure can be sustained. By contrast, we show that the proposed architecture can be scaled up to more servers and can still maintain the same level of reliability.

The second related study by Tewari *et al.* [15] investigated a clustered multimedia server architecture that also employs server-level striping. They used simulation and queueing models to analyze the QoS performance and to compare the cost-effectiveness of server-level striping with mirroring.

This paper differs from their study in two major ways. First, they employed different video distribution architecture in their study. Specifically, their system has two types of nodes: back-end nodes for storage, and front-end nodes for data delivery. Video data are striped across the back-end storage nodes while the front-end nodes assemble video data retrieved from the back-end storage nodes for delivery to video clients. This distribution architecture is called *independent proxy* [9]. By contrast, the architecture proposed in this paper has no intermediate delivery nodes—called *proxy-at-client* [9].

The primary advantage of the independent proxy architecture is client transparency: the details of the server cluster can be completely hidden because the client communicates with a single delivery node only. However, as the delivery nodes do not contribute to the system capacity, they will add to the cost of the system. Second, for economical reasons, a delivery node will likely serve many clients simultaneously. Hence, if a delivery node fails, services of all connected clients will be disrupted. The architecture proposed in this paper does not have this problem because no such delivery node is needed. Finally, given the rapid progress in CPU processing power, overhead in re-computing unavailable data due to server failures can readily be absorbed by the client CPU. Our experiments showed that even a low-end Pentium CPU can recompute lost data at a rate of more than 100 Mbps. Hence, performing the recomputations at the client not only better utilizes the client hardware, but also avoids potential bottlenecks at the intermediate delivery nodes.

Interested readers are referred to Lee [9] for more detailed comparisons of different parallel-server architectures.

The third related study is by Wong *et al.* [16]. Their RAIS architecture employed striping with distributed redundant units, which are computed from video data units. They proposed a special video transfer protocol to detect server failure. The client, armed with the redundant units and the survived data units, can then compute the lost units in real-time. In the simplest form, the redundant units are simply parity units, computed from exclusive-or between the video data units of the same stripe. This parity-based striping scheme can protect single-server failure and their protocol can maintain continuous video playback by means of additional buffering at the client.

This paper differs from RAIS [16] in three major ways. First, RAIS employs the client-pull service model where a client periodically sends requests to the servers to retrieve video blocks for playback. By contrast, the proposed architecture is based on the server-push service model where the servers continuously transmit data to a client. Briefly speaking, the two service models result in different designs for server and client, as well as different system requirements. Interested readers are referred to Lee [9] and Rao *et al.* [24] for comparisons between the two service models. Second, while RAIS employed block striping with distributed parity placement, we show that similar placement policy is not scalable for concurrent push. To solve this problem, we propose in Section IX a redundant server scheme that can be scaled up to any number of servers. Third, the RAIS study focused on implementation and experimentation. We establish in this paper a performance model for the proposed architecture to show that it is scalable to a larger number of servers and still can maintain the desired reliability.

III. The Concurrent-Push Architecture

We present an overview of the concurrent-push architecture in this section. Interested readers are referred to Lee [10] for more details. As shown in Fig. 1, the concurrent-push architecture is built on top of a cluster of homogenous servers, an admission scheduler, and the video clients. Each server is equipped with its own CPU, memory, disk storage, and network interface. The concurrent-push architecture defines the server striping policy, I/O scheduling policies, admission scheduling scheme, and client buffer management scheme.

Each server's storage space is divided into fixed-size units of Q bytes each. Figs. 2 and 3 depict the two striping policies in the concurrent-push architecture. We use N_S to denote the number of servers and N_C to denote the number of clients in the system. The one in Fig. 2 stripes video titles in fixed-size blocks of Q bytes, called block striping. The other one in Fig. 3, called sub-schedule striping scheme (SSS), stripes video titles in smaller units of U bytes, where $Q = N_S U$. Note that despite the difference in striping size, a disk transaction always retrieves a Q-byte block, albeit containing N_S stripe units instead of just one stripe unit in the case of SSS. SSS is developed to remove the client buffer requirement's dependency on N_S so that the system can be scaled up to more servers.

For I/O scheduling, each server employs a modified version of the group sweeping scheme (GSS): the asynchronous group

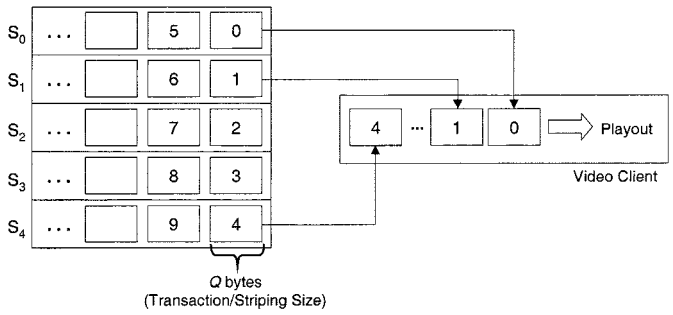

Fig. 2. Fixed-size block striping without redundancy.

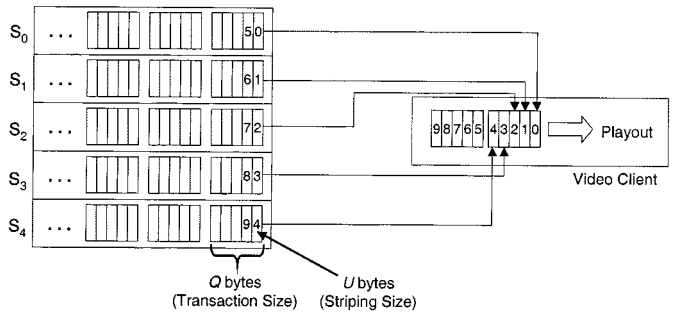

Fig. 3. Fixed-size sub-schedule striping without redundancy.

sweeping scheme (AGSS). The primary difference being that the assignment of a new video stream to a sweeping group is done externally by an admission scheduler in AGSS rather than internally by the server in GSS. This is necessary because server clocks are, in general, not precisely synchronized and hence server-based group assignment could become inconsistent, increasing transmission jitter. To start a new video session, a video client first sends a request to the admission scheduler. The admission scheduler will then schedule the servers to begin transmission to the client. Under concurrent-push, all servers transmit data to a client concurrently. With an average video bit-rate of R_V, each server will transmit at a reduced rate of R_V/N_S to maintain an aggregate data rate of R_V.

At the client side, it maintains a circular buffer comprising fixed-size blocks of Q bytes. A number of the buffer blocks are filled before video playback starts. These prefilled buffer blocks are used to prevent buffer underflow, while the remaining empty buffer blocks are used to prevent buffer overflow due to instantaneous variations in video-data consumption rate, network delay jitter, transmission jitter, etc. A detailed performance model on the concurrent-push architecture, including derivations on the buffer requirement and system response time, can be found in Lee [10].

IV. Redundancy Management

To support server-level fault tolerance, we need redundant data so that a client can recompute the unavailable video data after server failures. The problem of correcting data errors has been studied extensively in the literature. According to coding theory [25], one can encode a set of symbols with redundancies so that errors occurring within the set can be corrected later. However, server failure is slightly different in the sense

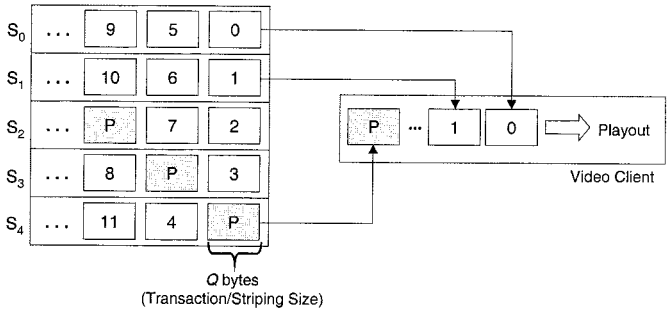

Fig. 4. Fixed-size block striping with redundancy of one.

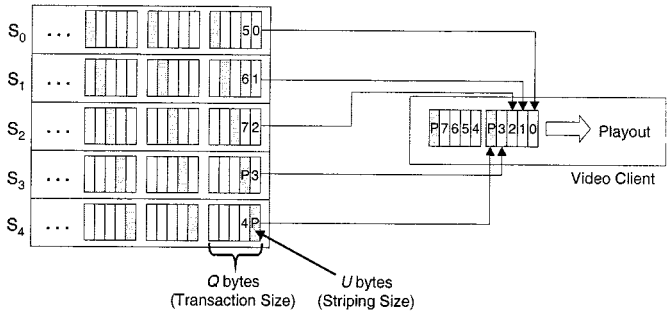

Fig. 5. Fixed-size sub-schedule striping with redundancy of one.

that there is really no error in the coding sense. Instead, a server failure introduces erasures—the absence of data.

Errors and erasures are different because for errors, data are still being received, but the content may be corrupted. In case of erasure, the expected data are simply missing, and hence no erroneous data will be received. Here we have implicitly assumed that the server is fail-stop, i.e., it stops sending out data upon failure. This type of failure could be caused by disk subsystem failure, network failure, power lost, or even software crashes. In any case, erasures are introduced into the video stream because data stored in the failed server will become unavailable.

According to coding theory, to recover an erased symbol (a unit of data) in a codeword (also called a parity group, or a stripe), one need to encode the data with at least one redundant symbol per codeword. One well-known coding algorithm called Reed–Solomon (RS) code [25] can encode data with any codeword size and level of redundancies. If one need to protect the system from only single-server failure, then an even simpler code—parity, can be used instead. For simplicity, we assume in this paper a generic code where each additional redundant symbol can recover one erasure.

Drawing related principles from RAID-5 [18], Figs. 4 and 5 depict the proposed redundant striping policies for block striping and sub-schedule striping. The basic idea is the same—introduces one or more redundant stripe units in every stripe. The redundant units are precomputed and distributed to the servers in a round-robin manner similar to a RAID-5 disk array. Note that a parity group spans all servers, and hence, the parity group size equals the number of servers in the system.

In [9], [16], two possible approaches are proposed for transmitting redundant data to the clients, namely FEC and ODC. These two schemes represent different tradeoffs: FEC simplifies system implementation and has lower client buffer requirement

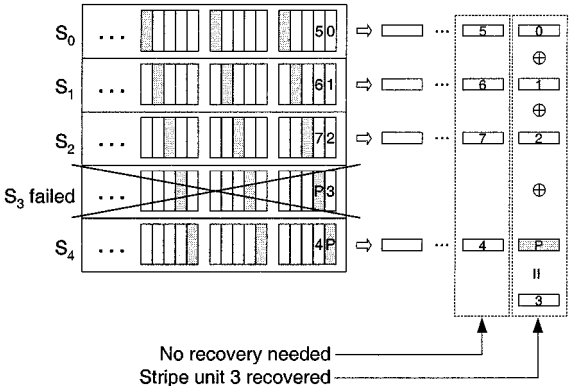

Fig. 6. Recovery of unavailable stripe units through erasure correction code.

and startup delay in certain cases, at the expense of network bandwidth overhead during normal operation (i.e., no failure), while ODC avoids this bandwidth overhead, at the expense of a more complicated system implementation and potentially larger buffer requirement and startup delay. We present a FEC-based transmission scheme for concurrent push in the next section, and a ODC-based transmission scheme in Section VI.

V. FEC

As the name suggests, servers under FEC transmit redundant data regardless of server failure. As redundant data are always received, the client can recompute unavailable data by erasure correction computation (see Fig. 6 for the case under sub-schedule striping). Hence, one does not need to detect server failure[1] for the sake of maintaining nonstop operation, and consequently system reconfiguration is also unnecessary. Clearly, this can greatly simplify the implementation and avoid other complications such as false alarm or undetected failure. The tradeoff is extra network bandwidth required to deliver redundant data during normal-mode operation. Specifically, with N_S servers and a redundancy level of K (i.e., up to K simultaneous server failures can be sustained), the network bandwidth overhead incurred will be given by

$$H_{\mathrm{FEC}} = \frac{K}{N_S - K} \tag{1}$$

For a small-scale system (i.e., N_S small) with high level of redundancy (i.e., K large), this overhead could become prohibitive. For example, with $N_S = 3$ and $K = 1$, the overhead would become 50%. Considering that a VoD system is expected to operate mostly in normal mode, this overhead may not be acceptable for systems with a small number of servers. The ODC scheme discussed next is designed to avoid this bandwidth overhead.

VI. ODC

Under ODC, the system does not transmit redundant data unless a server failure is detected, thereby avoiding the network bandwidth overhead incurred during normal-mode operation. In return, the system must detect server failures so that the system

[1]In practice, failure detection is still needed to notify the operator so that the failed server can be repaired or replaced.

can be reconfigured to start transmitting redundant data. This extra step of failure detection is not needed in FEC.

In theory, with N_S servers and a video data rate of R_V, each server only needs to transmit at a rate of R_V/N_S; we call it *Min-Rate* transmission. Upon a simultaneous x-server failure ($0 < x \leq K$), the surviving servers will have to increase the transmission rate from R_V/N_S to $R_V/(N_S - x)$ to maintain the same aggregate video bit-rate. This Min-Rate transmission scheme thus requires dynamic reconfiguration of the server scheduler as well as network bandwidth allocations. Alternatively, the system can maintain the transmission rate at $R_V(N_S - K)$; we call it *Std-Rate* transmission, even when there is no failure. The servers just skipped transmitting the redundant units. When a x-server failure occurs, the system will simply reconfigure x of the servers to start transmitting redundant data, thereby maintaining enough data for erasure correction at the clients. This approach eliminates the need to dynamically reconfigure the server scheduler and network connections.

If the network does not require per-channel resource allocation (e.g., FastEthernet), Min-Rate transmission will have no advantage over Std-Rate transmission, as the average rate is the same for both schemes. On the other hand, if the network requires per-channel resource allocation such as CBR service in ATM, then under Min-Rate transmission, the servers will need to re-negotiate a higher bandwidth allocation from the network upon detecting a failure. However, reconfiguring hundreds or even thousands of connections simultaneously could overload the network management center, which in turn could delay the reconfiguration process significantly. Therefore we conclude that the Min-Rate transmission scheme does not offer significant advantage over Std-Rate and is difficult to implement efficiently. By contrast, the Std-Rate transmission scheme is much simpler to implement, and so we will only consider the Std-Rate transmission scheme in the rest of the paper.

A. Failure-Detection Protocol

As discussed in the previous section, failure detection is necessary in ODC because redundant data are not normally transmitted. The goal is to detect a server failure quickly and accurately, so that the remaining servers can be reconfigured to begin transmitting redundant data. If the *detection delay*—defined as the time from a server fails to the time the remaining servers are notified of the failure, is too large then video playback hiccups can occur at the clients. On the other hand, the detection algorithm should not be overly sensitive in order to avoid false alarms. We propose an admission-scheduler-based (ASB) protocol for detecting server failures in this section.

In our previous investigations [8], [16], we found that incoming control requests could be delayed for a substantial amount of time (e.g., more than 1 s) due to intense I/O activities at the servers. Consequently, it would be more difficult to implement server-based fault-detection protocols that can quickly detect a failure. This motivates us to propose implementing fault-detection at the admission scheduler rather than at the servers. The admission scheduler is originally proposed to tackle the uneven group assignment problem arising from server clock jitters [10]. We propose extending the admission scheduler to simulate a video client. Unlike real video clients, however, received video data are simply discarded at the admission scheduler after bookkeeping is done, and the scheduler never performs any interactive control nor will the stream ever terminate (until system shutdown). At the servers, video data destined to the admission scheduler are not retrieved from the disks, but rather generated on-the-fly. Since the generated video data will not be interpreted at the admission scheduler, the server can avoid disk overhead by sending the same buffer repeatedly after updating header information such as stream offset or sequence number.

When a server fails, it simply stops transmitting data. Hence, a server failure can be inferred from the missing of video data at the admission scheduler. We assume that the admission scheduler is located close to the servers so that worst-case arrival deadlines are known for each and every video packets. Then the admission scheduler can declare a server to have failed if the arrival deadline is exceeded by a threshold of, say, T_{ASB} seconds. This threshold is introduced to reduce the possibility of false alarms caused by unexpected data delivery delays or occasional packet losses.

Note that the admission scheduler itself could also fail. However this type of failure will be less problematic because: 1) while new streams cannot be started, the failure will not affect existing streams and 2) compared to the video servers, the admission scheduler is much simpler and hence potentially far more reliable. For example, the admission scheduler can be diskless, so that disk failure can be avoided. ECC memory can be used to protect from memory faults, etc. We are currently investigating potential solutions such as replicated admission schedulers to tackle this final weak link.

B. Server Reconfiguration for Block Striping

Upon declaring that a server has failed, the admission scheduler will multicast a message to the surviving servers to notify them of the failure. The delay incurred will obviously be implementation dependent. For simplicity, we assume that the failure-detection delay is bounded and the maximum is given by T_D seconds. Upon receiving the failure notification, the servers will initiate a reconfiguration process to begin transmitting redundant blocks and to *retransmit* the necessary redundant blocks.

Fig. 7 depicts the scenario for reconfiguring a 5-server system under block striping. Note that we consider only one video stream for illustration and analysis, while in practice the same process occurs for all active video streams. All algorithms and procedures still apply and no modification is needed to extend to the multi-stream case. Note also that redundant video blocks are always retrieved, just not transmitted when there is no failure. One might notice that during normal operation, some disk bandwidth would then be wasted in retrieving redundant blocks that are not needed. It is conceivable that one can reuse this wasted bandwidth to serve extra video sessions during normal operation. However, these sessions will have to be disconnected upon server failure. More investigations are therefore needed to quantify the gains and the associated tradeoffs.

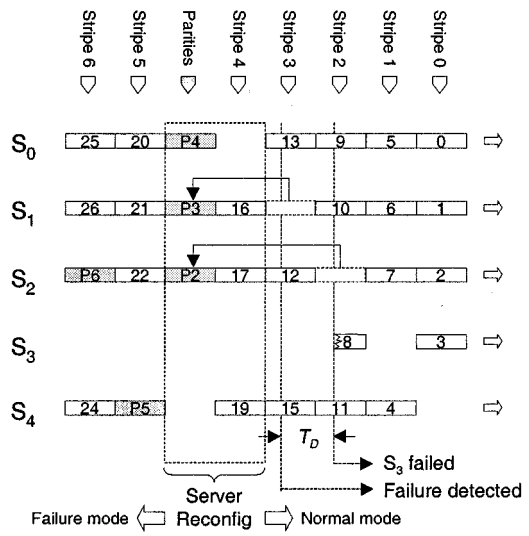

Fig. 7. Server reconfiguration under ODC with block striping.

Now assuming that failure occurs during service round j, then the servers will receive the failure notification latest by round

$$k = j + \left\lceil \frac{T_D}{T_F} \right\rceil \tag{2}$$

where T_F is the length of a service round defined in (5) below.

Knowing the failure, the servers will transmit redundant video blocks in addition to video blocks in the next round $(k+1)$ if there is one (e.g. P4 in Fig. 7). However, the $(k-j+1)$ stripes that are transmitted after the failure but before the failure is detected will have no redundant blocks transmitted (e.g., stripes 2 and 3). To enable the client to re-compute the lost stripe units, it is therefore necessary to *retransmit* the required redundant units (e.g., P2 and P3) for these stripes.

In a system with N_S servers and K redundant blocks per stripe, a maximum of $(k-j+1)K$ redundant blocks will have to be retransmitted. Note that this is the maximum because retransmission is not needed for lost redundant blocks. Assume the failure is a simultaneous K-server failure (worst-case scenario), leaving (N_S-K) working servers, the remaining servers can then retrieve and transmit (N_S-K) redundant units in a service round. Hence, a maximum of

$$N_R = \left\lceil \frac{(k-j+1)K}{(N_S-K)} \right\rceil = \left\lceil \frac{(\lceil T_D/T_F \rceil + 1)K}{(N_S-K)} \right\rceil \tag{3}$$

additional service rounds are required to retransmit the necessary redundant units. Consequently, transmission of subsequent stripes will be delayed by at most N_R rounds.

Now consider the recovery of stripe j (e.g. stripe 2 in Fig. 7). At the time (round k) the failure is detected, the current disk cycle is already retrieving stripe units for the next transmission cycle $k+1$. Hence, redundant units for stripe j can only be scheduled for retrieval in the next disk retrieval cycle, which in turn will be sent in transmission round $k+2$. Therefore, delivery of the redundant block required to recover stripe j will be delayed by

$$N_F = (k-j+2) = \left(\left\lceil \frac{R_V T_D}{Q(N_S-K))} \right\rceil + 2 \right) \tag{4}$$

service rounds. For a transmission rate of $R_V/(N_S-K)$ bytes/s under Std-Rate transmission, the time it takes to transmit a video block of Q bytes, i.e., length of a service round, is equal to

$$T_F = \frac{Q(N_S-K)}{R_V}. \tag{5}$$

Therefore, using (4) and (5), we can compute the delay for delivering the redundant block for stripe j from

$$D_F = N_F T_F = \left(\left\lceil \frac{R_V T_D}{Q(N_S-K))} \right\rceil + 2 \right) \frac{Q(N_S-K)}{R_V}. \tag{6}$$

Provided that $(N_S-K) \geq K$, (6) also bounds the delay for all stripes. To see why begins with (3)

$$\begin{aligned} N_R &= \left\lceil \frac{(k-j+1)K}{(N_S-K)} \right\rceil \\ &\leq (k-j+1), \quad \because ((N_S-K) \geq K) \\ &< (k-j+2) = N_F. \end{aligned} \tag{7}$$

This shows that the delay experienced by stripes $\{i | i > k+1\}$ transmitted after the failure is detected (N_R) is smaller than the delay experienced by stripe j (N_F). Therefore, the worst-case delay in (6) also bounds the delay for all stripes.

The additional delay will likely lead to video playback hiccups at the clients. If temporary service interruption can be tolerated, then the clients can simply suspend playback for D_F seconds to resynchronize with the new transmission schedule. Otherwise, we can introduce additional buffers at the client to sustain nonstop video playback during reconfiguration (Section VIII).

C. Server Reconfiguration for Sub-Schedule Striping

Fig. 8 depicts the server reconfiguration process for sub-schedule striping, with $N_S = 5$ and $K = 1$. Instead of considering service rounds, we consider micro-rounds, defined as the period for transmitting a stripe. Hence, a system with N_S servers will have N_S micro-rounds per service round. Note that in each service round, a server retrieves

$$Q_S = Q \frac{N_S}{(N_S-K)} \tag{8}$$

bytes of video data (instead of Q bytes in block striping) for every video stream and the length of a service round is

$$T_S = T_F \frac{N_S}{(N_S-K)} \tag{9}$$

seconds (instead of T_F seconds in block striping).

We assume that a K-server failure occurs during micro-round j and is detected in micro-round k. Similar to (2), we can obtain k from

$$k = j + \left\lceil \frac{N_S T_D}{T_S} \right\rceil. \tag{10}$$

Once notified of the failure, the servers will begin transmitting redundant units for subsequent stripes ($> k$). As each stripe

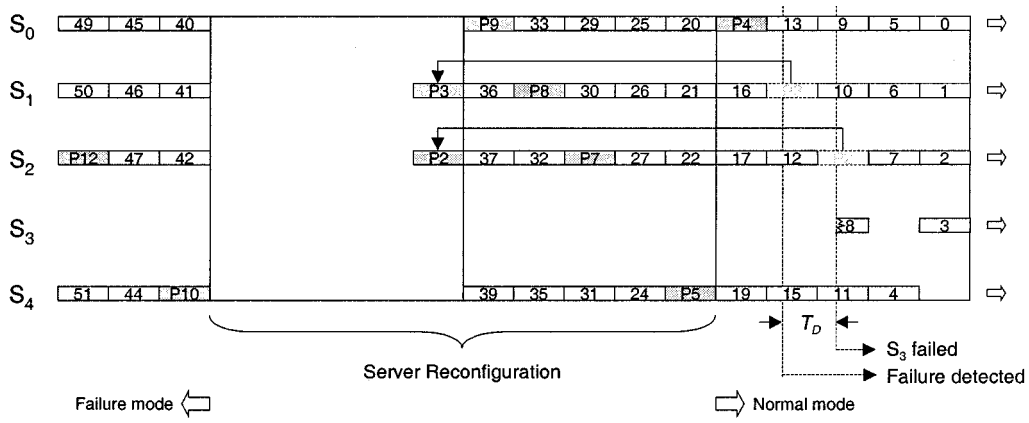

Fig. 8. Server reconfiguration under ODC with sub-schedule striping.

contains K redundant units, the system needs to retransmit up to $(k-j+1)K$ redundant units. This will require up to

$$n_R = \left\lceil \frac{(k-j+1)K}{(N_S-K)} \right\rceil = \left\lceil \frac{(\lceil (N_S-K)T_D/T_F \rceil + 1)K}{(N_S-K)} \right\rceil \tag{11}$$

micro-rounds for retransmitting the redundant units.

Note that this process has two subtle constraints. First, retransmission cannot start immediately in the next service round because the servers need another service round to retrieve the required redundant units. Second, even if $n_R < N_S$, the last service round for retransmission cannot be shortened because the disk requires a full service round to retrieve video blocks for transmission in the next round.

Similar to the block-striping case, the worst-case delay will be experienced by the stripe that is being transmitted when the failure occurs, provided that $(N_S-K) \geq K$. The worst-case delay can be up to

$$\begin{aligned} N_F &= k-j+(N_S-1)+N_S \\ &= \left\lceil \frac{(N_S-K)T_D}{T_F} \right\rceil + 2N_S - 1 \\ &= \left\lceil \frac{R_V T_D}{Q} \right\rceil + 2N_S - 1, \quad \because T_F = \frac{Q}{R_V}(N_S-K) \end{aligned} \tag{12}$$

micro-rounds, where $(k-j)$ is the worst-case delay due to failure detection, (N_S-1) is the worst-case delay to wait for the current service round to end, and N_S is the delay due to the first constraint discussed previously. Noting that the length of a micro-round is equal to T_S/N_S seconds, the delay is then given by

$$D_F = N_F \frac{T_S}{N_S} = \left(\left\lceil \frac{R_V T_D}{Q} \right\rceil + 2N_S - 1 \right) \frac{Q}{R_V} \tag{13}$$

seconds.

VII. Analysis of FEC

In this section, we derive the amount of client buffer needed to support fault tolerance under FEC so that nonstop playback can be sustained. Client buffers are originally introduced to absorb jitters in video-block playback times and delivery delays [10]. To support fault tolerance using FEC, we need additional client buffers to store a complete stripe (with redundant units) for erasure-correction computation. The derivations in the following sections are based on the model introduced in Lee [10]. The overall approach is to obtain upper and lower bounds for stripe unit arrival times and stripe unit consumption times. Then using the continuity condition, i.e., the latest arrival time for a stripe unit must not be later than the earliest consumption time, we can obtain the number of buffers required to prevent buffer underflow. We can obtain the number of buffers required to prevent buffer overflow in a similar way.

A. Buffer Requirement under Block Striping

We first consider the case for block striping. Let there be $L = (Y+Z)$ buffers (each Q bytes) at the client, organized as a circular buffer. Video playback starts once the first Y buffers are completely filled with video data. The client prefills the first Y buffers to prevent buffer underflow, and reserves the last Z buffers for incoming data to prevent buffer overflow.

Since all N_S servers transmit data to a client concurrently, the client will be receiving N_S video blocks simultaneously, of which (N_S-K) blocks containing video data and the rest containing redundant data. This suggests that Y must be multiples of N_S. Therefore, we consider groups of N_S buffers (i.e., group zero consists of blocks 0 to N_S-1, group one consists of blocks N_S to $2N_S-1$, and so on) and let $y = Y/N_S$ be the number of buffer groups prefilled.

Using techniques similar to Lee [10], we can obtain (see Appendix A.1 for derivations)

$$Y = \left\lceil 1 + \frac{\tau + f^+ - f^- - T_E}{(N_S-K)T_{\text{avg}}} \right\rceil N_S \tag{14}$$

for the number of buffers needed to prevent underflow, and

$$Z = \left\lceil 1 + \frac{\tau + f^+ - f^- + T_L}{(N_S-K)T_{\text{avg}}} \right\rceil N_S \tag{15}$$

for the number of buffers needed to prevent overflow. Note that T_E and T_L are jitter bounds for video block consumption, τ is the clock jitter among servers, and $f^+(f^+(f \geq 0)$ 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. See Table I for a summary of symbols and interested

TABLE I
SYSTEM PARAMETERS USED IN COMPUTING NUMERICAL RESULTS

System Parameters	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
Transmission time deviation	f^-, f^+	0ms
Server clock jitter	τ	100ms
Failure-detection delay	T_D	2s

readers are referred to Lee [10] for formal definitions of these parameters.

By setting $K = 0$ in (14) and (15), the equations reduce to the non-fault-tolerance version in Lee [10]. The total client buffer requirement is thus given by

$$B_{\text{FEC}}^{\text{BS}} = \left(2 + \left\lceil \frac{\tau + f^+ - f^- - T_E}{(N_S - K)T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L}{(N_S - K)T_{\text{avg}}} \right\rceil \right) N_S Q. \quad (16)$$

Note the independence of (16) from T_D as failure-detection and, consequently, server reconfiguration is not needed under FEC. However, we can also observe that the buffer requirement will increase when more servers are added to the system, suggesting that more buffers will be needed when scaling up the system.

B. Buffer Requirement under Sub-Schedule Striping

Under sub-schedule striping, each video block (Q_S bytes) at a server comprises multiple stripe units (U bytes each) and the size of a video block is given in (8). The client buffers now comprises $L = Y + Z$ buffer units of each Q_S bytes. Again we consider stripe units in groups of N_S units, i.e., group i comprises stripe units $\{iN_S, iN_S + 1, \ldots, (i+1)N_S - 1\}$. Then a group of N_S stripe units will correspond to exactly one buffer unit at the client. Using similar techniques (see Appendix A.2 for derivations), the buffer requirements can be found to be

$$Y = \left\lceil \frac{f^+ - f^- - T_E + \tau}{T_{\text{avg}}} \right\rceil + 1 \quad (17)$$

and

$$Z = \left\lceil \frac{f^+ - f^- + T_L + \tau}{T_{\text{avg}}} \right\rceil + 1. \quad (18)$$

Surprisingly, these are the same as the non-fault-tolerant case. This counter-intuitive result is explained by the fact that each group of buffers here has the size of Q_S bytes instead of Q bytes in the non-fault-tolerant case. Hence, the system does indeed need additional buffers to support fault tolerance and the total client buffer requirement is given by

$$B_{\text{FEC}}^{\text{SSS}} = \left(2 + \left\lceil \frac{\tau + f^+ - f^- - T_E}{T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L}{T_{\text{avg}}} \right\rceil \right) Q_S. \quad (19)$$

VIII. ANALYSIS OF ODC

Unlike FEC, ODC requires additional buffers to sustain continuous video playback during system reconfiguration. Incorporating this requirement, we derive the corresponding buffer requirement for block striping and sub-schedule striping in the following sections.

A. Buffer Requirement under Block Striping

Unlike FEC, a client operating under ODC will simultaneously receive $(N_S - K)$ instead of N_S video blocks. Therefore, a group of video blocks comprises only $(N_S - K)$ video blocks. Unlike FEC, derivations for the buffer requirements depend on whether the failure occurs before or after video playback starts. For the case where the failure occurs before video playback starts, the playback schedule will be delayed because playback cannot start until the required number of buffers are prefilled. The buffer requirements are found to be (see Appendix A.3 for derivations)

$$y_{\text{Before}} = \left\lceil 1 + \frac{\tau + f^+ - f^- - T_E}{(N_S - K)T_{\text{avg}}} \right\rceil \quad (20)$$

and

$$z_{\text{Before}} = \left\lceil 1 + \frac{\tau + f^+ - f^- + T_L + D_F}{(N_S - K)T_{\text{avg}}} \right\rceil. \quad (21)$$

For the case where the failure occurs after video playback starts, the playback schedule will not be affected. The buffer requirements are found to be

$$y_{\text{After}} = \left\lceil 1 + \frac{\tau + f^+ - f^- - T_E + D_F}{(N_S - K)T_{\text{avg}}} \right\rceil \quad (22)$$

and

$$z_{\text{After}} = \left\lceil 1 + \frac{\tau + f^+ - f^- + T_L}{(N_S - K)T_{\text{avg}}} \right\rceil. \quad (23)$$

Hence, the client buffer requirement is either

$$I_{\text{Before}} = y_{\text{Before}} + z_{\text{Before}} \quad (24)$$

or

$$I_{\text{After}} = y_{\text{After}} + z_{\text{After}} \quad (25)$$

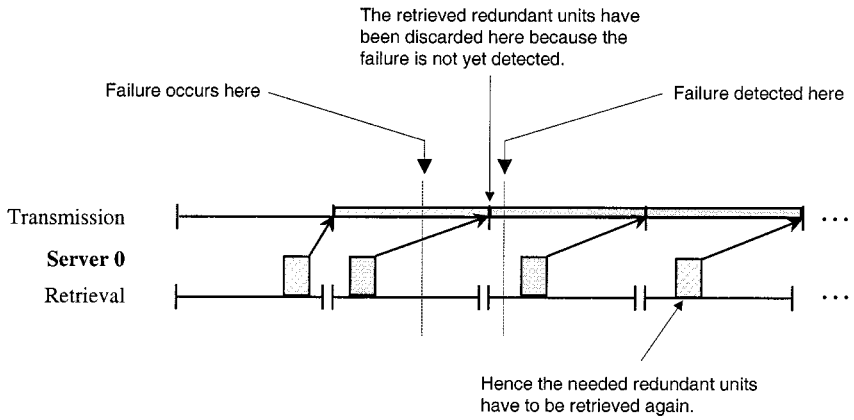

Fig. 9. The AGSS scheduler discards retrieved video blocks once transmission is completed.

whichever is larger. However, from (6): $D_F = N_F T_F$ and $T_F = (N_S - K)T_{\text{avg}}$. Therefore, the two equations are in fact equivalent. The total client buffer requirement is thus given by

$$B_{\text{ODC}}^{\text{BS}} = \left(2 + N_F + \left\lceil \frac{\tau + f^+ - f^- - T_E}{(N_S - K)T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L}{(N_S - K)T_{\text{avg}}} \right\rceil\right)(N_S - K)Q. \quad (26)$$

B. Buffer Requirement under Sub-Schedule Striping

To derive the client buffer requirement for sub-schedule striping, we again consider stripe units in groups of $(N_S - K)$, i.e., group i comprises stripe units $\{i(N_S - K), i(N_S - K) + 1, \ldots, (i+1)N_S - K) - 1)\}$. Now unlike FEC, each group of stripe units has the size of Q bytes, instead of Q_S bytes under FEC. Hence, the client buffer comprises $L = Y + Z$ buffer units, each of size Q bytes. Proceeding the derivations in the same manner (see Appendix A.4 for details), we can obtain the total buffer requirements from

$$B_{\text{ODC}}^{\text{SSS}} = \left(2 + N_F + \left\lceil \frac{\tau + f^+ - f^- - T_E}{T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L}{T_{\text{avg}}} \right\rceil\right)Q. \quad (27)$$

From (12), we can see that N_F is proportional to N_S. This implies that the buffer requirement is also proportional to N_S. As sub-schedule striping is originally proposed [10] to maintain a constant client buffer requirement independent of system scale (i.e., N_S), the extension to ODC appears to have defeated this goal. We propose a redundant server scheme in the next section to tackle this problem.

IX. Redundant Server Scheme

A closer look at Fig. 8 reveals why buffer requirement increases with system scale under ODC. First, retransmission of redundant stripe units cannot start in the current service round. This incurs a worst-case delay equal to $(N_S - 1)T_{\text{avg}}$ seconds, which obviously is proportional to the system scale. Second, retransmissions cannot start even in the next service round due to the need to retrieve redundant stripe units, incurring another delay of $N_S T_{\text{avg}}$ seconds, which again is proportional to the system scale.

The key to the previous two observations is in the server scheduler. First, under the AGSS scheduler [10], redundant units are discarded together with the video data units once the service round ends to allow buffer reuse. Hence if the failure-detection period spans two service rounds as shown in Fig. 9, redundant units for the previous round will have been discarded by the time the failure is detected, rendering immediate retransmission of redundant stripe units impossible.

To tackle this problem, one can modify the AGSS scheduler such that redundant units are retained longer to cater for server failure. However, we propose a redundant server scheme (RSS) to store all redundant units centrally in one or more (K to be exact) *redundant* servers instead of distributing them over all servers. RSS has three advantages over simply increasing the buffer holding time in AGSS.

First, RSS requires only the redundant servers, instead of all servers, to have the additional memory to buffer redundant units. Therefore, the total server buffer requirement is reduced. Second, redundant units can be stored continuously on the disks in the redundant servers such that retrievals are much more efficient. By contrast, redundant units in the original distributed scheme are scattered on the disk and hence a separate disk I/O is required to retrieve each redundant unit. Third, under RSS, retransmission of the redundant units can start as soon as the failure is detected, without the need to wait for the current stripe unit to complete transmission. This is possible because the redundant servers are idle before a failure is detected.

Assume failure occurs at time t_f during the transmission of stripe j, then it will be detected latest by time $(t_f + T_D)$. Since retransmission of redundant stripe units can start immediately upon failure detection, as shown in Fig. 10, the required redundant unit will be transmitted by time $(t_f + T_D + T_{\text{avg}})$. Now, let t_j be the time for which transmission of stripe j ends. Then, it is easy to see that

$$t_f \le t_j \le t_f + T_{\text{avg}}. \quad (28)$$

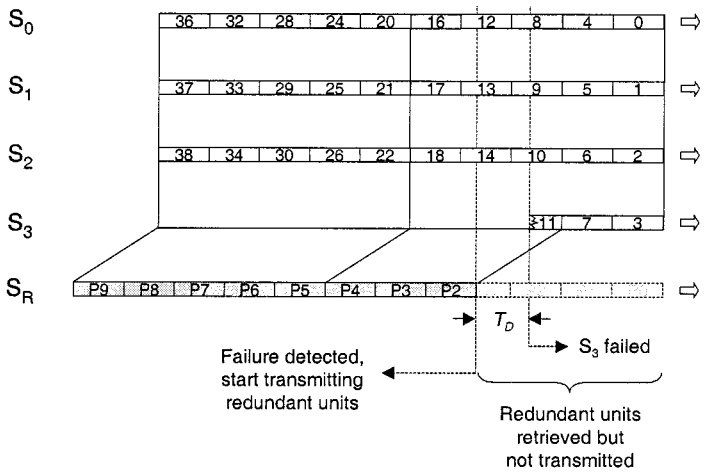

Fig. 10. Transmission scenario for the redundant server scheme.

Since the client will need to wait for the redundant unit before stripe j can be recomputed, the delay incurred in receiving stripe j will be given by

$$\begin{aligned}(t_f + T_D + T_{\text{avg}}) - t_j &\le (t_f + T_D) - t_f \\ &= T_D + T_{\text{avg}} \end{aligned} \tag{29}$$

which, finally, is independent of the system scale. Using derivations similar to Section VIII, we can obtain the client buffer requirement from

$$B_{\text{ODC}}^{\text{RSS}} = \left(3 + \left\lceil \frac{\tau + f^+ - f^- - T_E + T_D}{T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L}{T_{\text{avg}}} \right\rceil\right) Q \tag{30}$$

for the case where failure occurs after playback have begun, and

$$B_{\text{ODC}}^{\text{RSS}} = \left(3 + \left\lceil \frac{\tau + f^+ - f^- - T_E}{T_{\text{avg}}} \right\rceil + \left\lceil \frac{\tau + f^+ - f^- + T_L + T_D}{T_{\text{avg}}} \right\rceil\right) Q \tag{31}$$

for the case where failure occurs before playback begins.

To support immediate retransmission of redundant units, the redundant servers will need to retain redundant units longer than in the original AGSS scheduler. In particular, the server will need to keep retrieved redundant units (in blocks of N_S units) for

$$\left\lceil \frac{T_D}{N_S T_{\text{avg}}} \right\rceil + 1 \tag{32}$$

service rounds (instead of one round in AGSS). Hence, the buffer requirement for the redundant servers will be given by

$$B_{\text{server}} = Q N_S \Lambda \left(1 + \frac{1}{G} + \left\lceil \frac{T_D}{N_S T_{\text{avg}}} \right\rceil\right) \tag{33}$$

where Λ is the client-server ratio and G is the number of groups per service round [10].

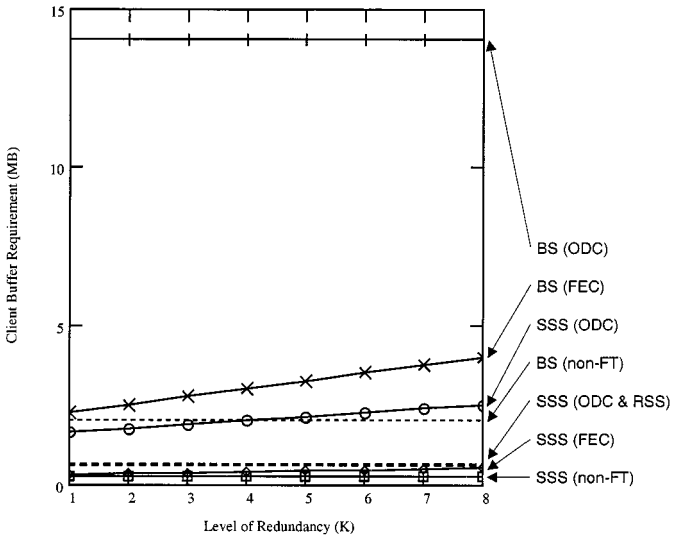

Fig. 11. Client buffer requirement versus level of redundancy.

X. Numerical Results

Based on the performance models derived in the previous sections, we compute and present numerical results in this section to illustrate the system resource requirement under various scenarios and study the sensitivity to key system parameters. Table I lists the values for the 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 [8].

A. Buffer Requirement versus Level of Redundancy

Fig. 11 plots the client buffer requirement versus the level of redundancy. There are a total of $(8 + K)$ servers in the system. There are two observations. First, sub-schedule striping in general requires less client buffer than block striping. Second, sub-schedule striping with ODC and RSS is the only scheme that has constant client buffer requirement irrespective of redundancy level. Even the buffer requirement for the FEC case increases with K. This is explained by the fact that under FEC, the client must receive and process video data in parity groups. Hence,

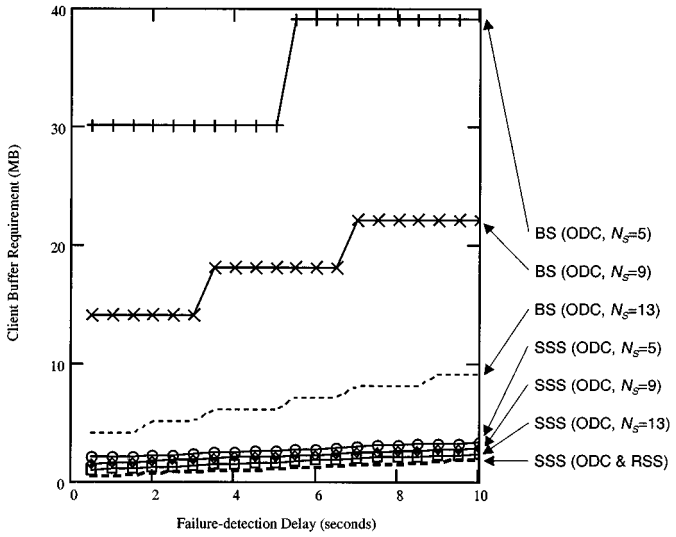

Fig. 12. Client buffer requirement versus failure-detection delay.

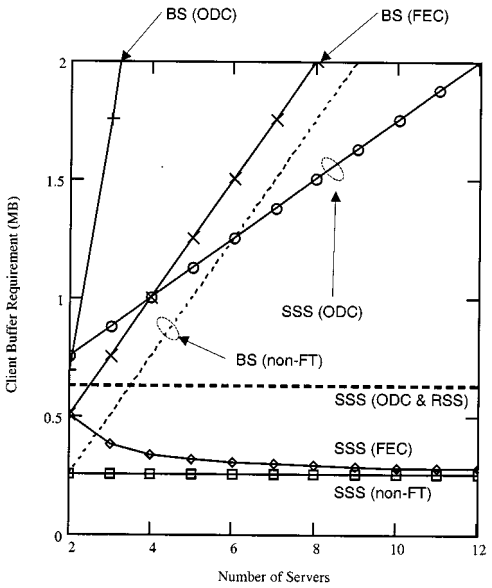

Fig. 13. Client buffer requirement versus number of servers.

as K increases, so do the parity group size and, consequently, the buffer requirement. By contrast, redundant stripe units are not transmitted until failure is detected under ODC with RSS. Therefore, the buffer requirement does not depend on K at all.

B. Buffer Requirement versus Failure-detection Delay

Fig. 12 studies the sensitivity of buffer requirement with respect to failure-detection delay for various ODC system configurations. FEC is not plotted because the buffer requirement is independent from the failure-detection delay. For all cases in Fig. 12, the buffer requirement increases with longer failure-detection delay. The results show that sub-schedule striping again achieves lower buffer requirement in general, with ODC/RSS achieving the smallest buffer requirement.

C. Buffer Requirement versus System Scale

Fig. 13 plots the client buffer requirement versus the number of servers in the system (i.e. system scale). The level of redundancy is one (i.e., $K = 1$) and the failure-detection delay is 2 s. The first result is that block striping is nonscalable. This extends the results in Lee [10] for the non-fault-tolerant case to FEC and ODC. The second result is that sub-schedule striping with ODC is also nonscalable, although the slope is smaller than block striping. Finally, we can observe that only sub-schedule striping under FEC, and under ODC with RSS are scalable, the latter being completely independent of the system scale. Interestingly, buffer requirements under FEC decreases for more servers and approaches the non-fault-tolerant case. This is because the level of redundancy is fixed and hence the redundancy overhead incurred decreases when more servers are added.

D. Scalability under Constant Reliability

The previous results are obtained with constant redundany level. However, the system-level reliability will inevitably decrease with more servers if the redundancy level is kept constant. To the service provider, it would be desirable to obtain the system requirement under a given minimum system-level reliability. Formally, we assume that the servers are homogeneous

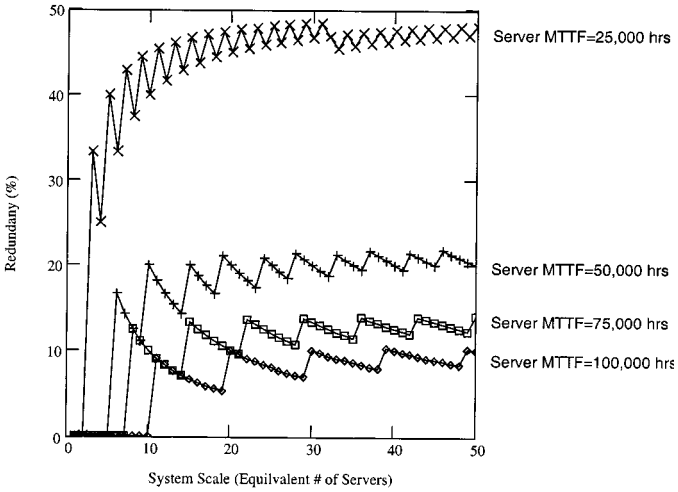

Fig. 14. Proportion of redundancy versus system scale with system MTTF of 10 000 hours.

with an exponentially-distributed life time. Let V be the MTTF for a server node, and let W be the MTTF for the system. Then given n, the desired system capacity, defined as $n = (N_S - K)$, and V, we can obtain W from [26]

$$W(n,K) = V \sum_{i=n}^{n+K} \frac{1}{i}. \tag{34}$$

Next, we can determine K such that the system MTTF is equal to or larger than a minimum, denoted by $W_{\min}$

$$K = \min\{k | W(n,k) \geq W_{\min}, k = 0, 1, \ldots\}. \tag{35}$$

Fig. 14 plots the amount of redundancy required versus the system scale under a system MTTF requirement of 10 000 hours. The results suggest that the amount of redundany

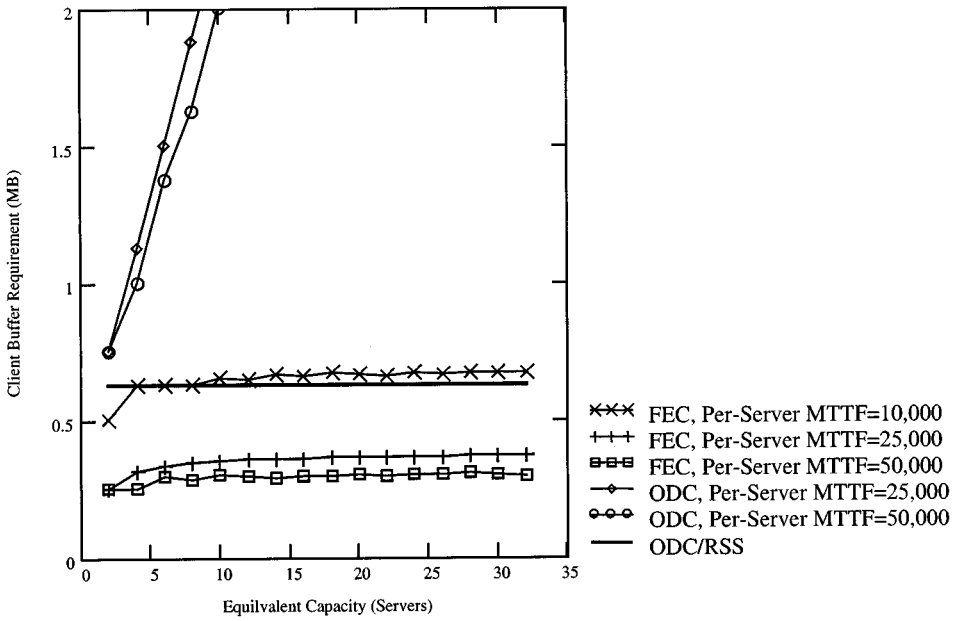

Fig. 15. Client buffer requirement versus system scale with system MTTF of 10 000 hrs.

required depends primary on the server MTTF and not on the system scale.

Once K is known, we can obtain the client buffer requirement accordingly. The results are plotted in Fig. 15, with the vertical axis representing client buffer requirement and the horizontal axis representing n. The minimum system MTTF is fixed at 10 000 hours. The results show that sub-schedule striping with both ODC/RSS and FEC are scalable under the given system MTTF constraint. On the other hand, we note that while the buffer requirement for FEC depends on the server MTTF, the buffer requirement for ODC/RSS is constant and independent of server MTTF. We also note that FEC requires less buffer if server MTTF is significantly larger than system MTTF (e.g., for cases with server MTTF = 25 000 and 50 000 hours). Otherwise (e.g. server MTTF = 10 000 hours) ODC/RSS will achieve a lower buffer requirement.

XI. Conclusion

In this paper, we investigate various protocols and algorithms to support server-level fault tolerance in the concurrent push architecture. In particular, we propose and compare two fault-tolerant protocols, FEC and ODC, and two striping policies, block striping and sub-schedule striping. The first result is that FEC is simpler in implementation, does not require failure detection, and is inherently scalable to any number of servers under sub-schedule striping. The only downside is additional network bandwidth overhead during normal operation. Surprisingly, analytical results show that ODC is not scalable if redundant data is distributed over all servers (similar to RAID-5 in disk arrays), even with sub-schedule striping. To tackle this problem, we propose storing redundant data centrally in redundant servers to avoid the reconfiguration delay. We increase the buffer holding time at the redundant servers to enable quick redundant data transmission. Analytical results prove that this redundant server scheme enables ODC to become scalable to any number of servers. Finally, we compute numerical results to show the feasibility of the proposed architecture under real-world conditions. With the proposed architecture, a concurrent-push-based parallel video server will be able to sustain multiple simultaneous-server failure and yet, can maintain nonstop continuous video playback for all clients.

Appendix

A. Derivations of Buffer Requirement for Block Striping Under FEC

Among the N_S servers, assume the earliest transmission for the first round starts at time t_0, then the last transmission for the first round must start at the latest by time $t_0 + \tau$, where τ is the clock jitter among servers. The time for video block group i to be completely filled, denoted by $F(i)$, is therefore bounded by [10]

$$((i+1)T_F+t_0+f^-) \le F(i) \le ((i+1)T_F+t_0+\tau+f^+) \quad (36)$$

where T_F is as given in (4) and f^+ ($f^+ \ge 0$) and f^- ($f^- \le 0$) 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 equal to $F(y-1)$. Hence, the playback time for video block group i, denoted by $P(i)$, is bounded by

$$\begin{aligned}&(iN_ST_{\text{avg}} + F(y-1) + T_E)\\ &\quad \le P(i) \le (iN_ST_{\text{avg}} + F(y-1) + T_L)\end{aligned} \quad (37)$$

where

$$T_{\text{avg}} = \frac{Q}{R_V} \quad (38)$$

is the average playback time for one video block, and T_E, T_L are the jitter bounds for video-block consumption variations [10].

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

$$\max\{F(i)\} \le \min\{P(i)\}. \tag{39}$$

For the LHS, noting that $T_F = (N_S - K)T_{\text{avg}}$ (c.f. (5) and (38)), we then have

$$\max\{F(i)\} = (i+1)(N_S - K)T_{\text{avg}} + t_0 + \tau + f^+. \tag{40}$$

Similarly, the RHS is

$$\begin{aligned}\min\{P(i)\} &= iN_S T_{\text{avg}} + \min\{F(y-1)\} + T_E \\ &= iN_S T_{\text{avg}} + y(N_S - K)T_{\text{avg}} + t_0 + f^- + T_E\end{aligned} \tag{41}$$

Substituting (40), (41) into (39) gives

$$\begin{aligned}&((i+1)(N_S - K)T_{\text{avg}} + t_0 + \tau + f^+) \\ &\quad \le (iN_S T_{\text{avg}} + y(N_S - K)T_{\text{avg}} + t_0 + f^- + T_E).\end{aligned} \tag{42}$$

Rearranging, we can then obtain y

$$y \ge 1 + \frac{f^+ - f^- - T_E + \tau}{(N_S - K)T_{\text{avg}}}. \tag{43}$$

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

$$Y = \left\lceil 1 + \frac{\tau + f^+ - f^- - T_E}{(N_S - K)T_{\text{avg}}} \right\rceil N_S. \tag{44}$$

On the other hand, to guarantee that the client buffer will not be overflowed by incoming video data, we need to ensure that the ith video block group starts playback before the $(i+l-2)$th video block group is completely received, where $l = L/N_S$. This is because the client buffers are organized as a circular buffer, and we must always have at least one group of N_S free buffers to receive 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

$$\min\{F(i+l-2)\} \ge \max\{P(i)\}. \tag{45}$$

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

$$Z = \left\lceil 1 + \frac{\tau + f^+ - f^- + T_L}{(N_S - K)T_{\text{avg}}} \right\rceil N_S. \tag{46}$$

B. Derivations of Buffer Requirement for Sub-Schedule Striping Under FEC

The filling time for group i of a video stream started at time t_0 is bounded by

$$\begin{aligned}&((i+1)T_{\text{avg}} + t_0 + f^-) \\ &\quad \le F(i) \le ((i+1)T_{\text{avg}} + t_0 + f^+ + \tau).\end{aligned} \tag{47}$$

Since the client starts video playback after filling the first Y groups of buffers, the playback time for video block group 0 is simply equal to $F(Y-1)$. Hence, the playback time for video block group i, denoted by $P(i)$, is bounded by

$$\begin{aligned}&(iT_{\text{avg}} + F(Y-1) + T_E) \\ &\quad \le P(i) \le (iT_{\text{avg}} + F(Y-1) + T_L).\end{aligned} \tag{48}$$

Substituting the upper bound of (47) and the lower bound of (48) into the continuity condition in (39) gives

$$\begin{aligned}&((i+1)T_{\text{avg}} + t_0 + f^+ + \tau) \\ &\quad \le (iT_{\text{avg}} + \min\{F(Y-1)\} + T_E)\end{aligned} \tag{49}$$

or

$$\begin{aligned}&((i+1)T_{\text{avg}} + t_0 + f^+ + \tau) \\ &\quad \le (iT_{\text{avg}} + (YT_{\text{avg}} + t_0 + f^-) + T_E).\end{aligned} \tag{50}$$

Rearranging, we can obtain Y from

$$Y = \left\lceil \frac{f^+ - f^- - T_E + \tau}{T_{\text{avg}}} \right\rceil + 1. \tag{51}$$

Using similar derivations, we can obtain Z from

$$Z = \left\lceil \frac{f^+ - f^- + T_L + \tau}{T_{\text{avg}}} \right\rceil + 1. \tag{52}$$

C. Derivations of Buffer Requirement for Block Striping Under ODC

Assume that a failure occurs during the transmission of group j, then for those groups received before a failure (i.e., $0 \le i < j$), the filling time is bounded by

$$\begin{aligned}&((i+1)T_F + t_0 + f^-) \\ &\quad \le F_N(i) \le ((i+1)T_F + t_0 + \tau + f^+), \\ &\qquad 0 \le i < j\end{aligned} \tag{53}$$

However, groups transmitted after the failure $(i \ge j)$ will be deferred due to server reconfiguration. According to Section VI-B, the worst-case delay due to reconfiguration is D_F seconds. Hence, the maximum filling time is bounded by

$$\begin{aligned}&((i+1)T_F + t_0 + f^- + D_F) \\ &\quad \le F_F(i) \le ((i+1)T_F + t_0 + \tau + f^+ + D_F), \\ &\qquad i \ge j.\end{aligned} \tag{54}$$

Merging (53) and (54) gives bounds for the general case

$$\begin{aligned}&((i+1)T_F + t_0 + f^-) \\ &\quad \le F(i) \le ((i+1)T_F + t_0 + \tau + f^+ + D_F), \quad \forall i.\end{aligned} \tag{55}$$

The bounds for $P(i)$ depend on whether a failure occurs before or after playback has begun. Specifically, if a failure occurs before playback begins, then playback will be delayed up to D_F

seconds due to the need to reconfigure the servers to complete the prefill process

$$(i(N_S-K)T_{\text{avg}}+F_F(y-1)+T_E) \le P_{\text{Before}}(i) \le (i(N_S-K)T_{\text{avg}}+F_F(y-1)+T_L). \quad (56)$$

Otherwise, if the failure occurs after playback has begun, then the playback schedule will not be affected

$$(i(N_S-K)T_{\text{avg}}+F_N(y-1)+T_E) \le P_{\text{After}}(i) \le (i(N_S-K)T_{\text{avg}}+F_N(y-1)+T_L) \quad (57)$$

Now if the failure occurs before playback begins, then invoking the continuity condition gives

$$\max\{F(i)\} \le \min\{P_{\text{Before}}(i)\} \quad (58)$$

or

$$((i+1)T_F+t_0+\tau+f^+ +D_F) \le (i(N_S-K)T_{\text{avg}}+\min\{F_F(y-1)\}+T_E) \quad (59)$$

Substituting the lower bound of (54) into (59) and noting $T_F = (N_S-K)T_{\text{avg}}$, we get

$$((i+1)(N_S-K)T_{\text{avg}}+t_0+\tau+f^+ +D_F) \le (i(N_S-K)T_{\text{avg}}+(y(N_S-K)T_{\text{avg}} +t_0+f^- +D_F)+T_E). \quad (60)$$

Rearranging, we can obtain y from

$$y_{\text{Before}} = \left\lceil 1+\frac{\tau+f^+ -f^- -T_E}{(N_S-K)T_{\text{avg}}} \right\rceil. \quad (61)$$

Similarly, if the failure occurs after playback has begun, then the continuity condition becomes

$$\max\{F(i)\} \le \min\{P_{\text{After}}(i)\} \quad (62)$$

or

$$((i+1)T_F+t_0+\tau+f^+ +D_F) \le (i(N_S-K)T_{\text{avg}}+\min\{F_N(y-1)\}+T_E). \quad (63)$$

Solving, we can obtain y from

$$y_{\text{After}} = \left\lceil 1+\frac{\tau+f^+ -f^- -T_E+D_F}{(N_S-K)T_{\text{avg}}} \right\rceil. \quad (64)$$

Similarly for Z, we also need to consider the two cases. For first the case where failure occurs before playback begins, we have

$$\min\{F(i+l-2)\} \ge \max\{P_{\text{Before}}(i)\} \quad (65)$$

or

$$((i+l-1)(N_S-K)T_{\text{avg}}+t_0+f^-) > (i(N_S-K)T_{\text{avg}}+\max\{F_F(y-1)\}+T_L). \quad (66)$$

Substituting the upper bound of (54) into (66), we have

$$((i+l-1)(N_S-K)T_{\text{avg}}+t_0+f^-) > (iN_ST_{\text{avg}}+(y(N_S-K)T_{\text{avg}} +t_0+\tau+f^+ +D_F)+T_L). \quad (67)$$

Rearranging, we can obtain $z=(l-y)$ as

$$z_{\text{Before}} = \left\lceil 1+\frac{\tau+f^+ -f^- +T_L+D_F}{(N_S-K)T_{\text{avg}}} \right\rceil. \quad (68)$$

For the second case, playback is not delayed by the failure. Hence, we have

$$\min\{F(i+l-2)\} \ge \max\{P_{\text{After}}(i)\} \quad (69)$$

or

$$((i+l-1)(N_S-K)T_{\text{avg}}+t_0+f^-) > (i(N_S-K)T_{\text{avg}}+\max\{F_N(y-1)\}+T_L). \quad (70)$$

Substituting the upper bound of (53) into (70), we have

$$((i+l-1)(N_S-K)T_{\text{avg}}+t_0+f^-) > (iN_ST_{\text{avg}}+(y(N_S-K)T_{\text{avg}}+t_0+\tau+f^+)+T_L). \quad (71)$$

Rearranging, we can obtain $z=(l-y)$ as

$$z_{\text{After}} = \left\lceil 1+\frac{\tau_f^+ -f^- +T_L}{(N_S-K)T_{\text{avg}}} \right\rceil. \quad (72)$$

D. Derivations of Buffer Requirement for Sub-Schedule Striping Under ODC

Assuming failure occurs during transmission of group j, then the filling time for group i of a video stream started at time t_0 is bounded by

$$((i+1)T_{\text{avg}}+t_0+f^-) \le F_N(i) \le ((i+1)T_{\text{avg}}+t_0+f^+ +\tau), \quad 0 \le i < j \quad (73)$$

$$((i+1)T_{\text{avg}}+t_0+f^- +D_F) \le F_F(i) \le ((i+1)T_{\text{avg}}+t_0+f^+ +\tau+D_F), \quad i \ge j. \quad (74)$$

Merging (73) and (74) gives the universal bounds for $F(i)$

$$((i+1)T_{\text{avg}}+t_0+f^-) \le F(i) \le ((i+1)T_{\text{avg}}+t_0+f^+ +\tau+D_F), \quad \forall i. \quad (75)$$

Similarly, the playback schedule is bounded by

$$(iT_{\text{avg}}+F_F(Y-1)+T_E) \le P_{\text{Before}}(i) \le (iT_{\text{avg}}+F_F(Y-1)+T_L) \quad (76)$$

for the case where failure occurs before playback begins, and

$$(iT_{\text{avg}} + F_N(Y-1) + T_E) \leq P_{\text{After}}(i) \leq (iT_{\text{avg}} + F_N(Y-1) + T_L) \quad (77)$$

for the case where failure occurs after playback has begun.

Invoking the continuity condition, we can obtain the corresponding bounds for Y as follows:

$$Y_{\text{Before}} = \left\lceil \frac{f^+ - f^- - T_E + \tau}{T_{\text{avg}}} \right\rceil + 1 \quad (78)$$

$$Y_{\text{After}} = \left\lceil \frac{f^+ - f^- - T_E + \tau + D_F}{T_{\text{avg}}} \right\rceil + 1. \quad (79)$$

Using similar derivations, we can obtain the corresponding bounds for Z as follows:

$$Z_{\text{Before}} = \left\lceil \frac{f^+ - f^- + T_L + \tau + D_F}{T_{\text{avg}}} \right\rceil + 1 \quad (80)$$

$$Z_{\text{After}} = \left\lceil \frac{f^+ - f^- + T_L + \tau}{T_{\text{avg}}} \right\rceil + 1. \quad (81)$$

ACKNOWLEDGMENT

The authors would like to express their gratitude to the anonymous reviewers for their insightful comments and suggestions in improving this paper.

REFERENCES

[1] R. Buck, "The Oracle Media Server for nCube Massively Parallel Systems," in *Proc. 8th Int. Parallel Processing Symp.*, 1994, pp. 670–673.

[2] H. Taylor, D. Chin, and S. Knight, "The magic video-on-demand server and real-time simulation system," *IEEE Parallel Distrib. Technol.: Syst. and Applic.*, vol. 3, pp. 40–51, 1995.

[3] 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*, 1996, pp. 372–381.

[4] C. Bernhardt and E. Biersack, "The server array: A scalable video server architecture," in *High-Speed Networks for Multimedia Applications*. Norwell, MA: Kluwer, 1996.

[5] W. J. Bolosky, J. S. Barrera, III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhyvold, and R. F. Rashid, "The tiger video fileserver," in *Proc. 6th Int. Workshop Network and Operating System Support for Digital Audio and Video*, Zushi, Japan, Apr. 1996.

[6] M. M. Buddhikot and G. M. Parulkar, "Efficient data layout, scheduling and playout control in MARS," in *Proc. NOSSDAV'95*, Berlin, Germany, 1995, pp. 318–329.

[7] C. S. Freedman and D. J. DeWitt, "The SPIFFI scalabale video-on-demand system," in *Proc. ACM SIGMOD'95*, New York, June 1995, pp. 352–363.

[8] 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*, 1996, pp. 27–34.

[9] Y. B. Lee, "Parallel video servers—A tutorial," *IEEE Multimedia*, vol. 5, pp. 20–28, June 1998.

[10] ——, "Concurrent push—A scheduling algorithm for push-based parallel video servers," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 9, Apr. 1999.

[11] Y. B. Lee and P. C. Wong, "Performance analysis of a pull-based parallel video server," IEEE Trans. Parallel Distrib. Syst., vol. 11, Dec. 2000, to be published.

[12] P. Lougher, D. Pegler, and D. Shepherd, "Scalable storage servers for digital audio and video," in *Proc. Inst. Elect. Eng. Int. Conf. Storage and Recording Systems 1994*, London, U.K., 1994, pp. 140–143.

[13] 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.

[14] R. Tewari, R. Mukherjee, and D. M. Dias, "Real-time issues for clustered multimedia servers,", IBM Res. Rep. RC20020, June 1995.

[15] R. Tewari, D. M. Dias, R. Mukherjee, and H. M. Vin, "High availability in clustered multimedia servers," in *Proc. 12th Int. Conf. Data Engineering*, 1996, pp. 645–654.

[16] P. C. Wong and Y. B. Lee, "Redundant array of inexpensive servers (RAIS) for on-demand multimedia services," in *Proc. ICC'97*, 1997, pp. 787–792.

[17] M. Wu and W. Shu, "Scheduling for large-scale parallel video servers," in *Proc. 6th Symp. Frontiers of Massively Parallel Computation*, 1996, pp. 126–133.

[18] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, "Introduction to redundant array of inexpensive disks (RAID)," in *Proc. COMPCON Spring '89*, 1989, pp. 112–117.

[19] S. Berson and L. Golubchik, "Fault tolerant design of multimedia servers," in *Proc. SIGMOD'95*, pp. 364–375.

[20] F. A. Tobagi, J. Pang, R. Baird, and M. Gang, "Streaming RAID™—A disk array management system for video files," in *Proc. ACM Multimedia*, Anaheim, CA, 1995, pp. 393–400.

[21] L. Golubchik and R. Muntz, "Designing efficient fault tolerant VOD storage servers: techniques, analysis, and comparison," in *Proc. SPIE'95*, 1989, pp. 112–117.

[22] H. M. Vin, P. J. Shenoy, and S. S. Rao, "Efficient failure recovery in multi-disk multimedia servers," in *Proc. 25th Ann. Symp. Fault Tolerant Computing (FTCS'95)*, Pasadena, CA, pp. 12–21.

[23] A. Merchant and P. S. Yu, "Design and modeling of clustered RAID," in *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, 1992, pp. 140–49.

[24] S. S. Rao, H. M. Vin, and A. Tarafdar, "Comparative evaluation of server-push and client-pull architectures for multimedia servers," in *Proc. 6th NOSSDAV*, Zushi, Japan, Apr. 1996, pp. 45–48.

[25] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995, pp. 227–34.

[26] C. E. Ebeling, *An Introduction to Reliability and Maintainability Engineering*. New York: McGraw-Hill, 1997, p. 89.

Jack Y. B. Lee is an Assistant Professor in the Department of Information Engineering, Chinese University of Hong Kong. His research interests include distributed multimedia systems, fault-tolerant systems, and Internet computing.