# Parallel Video Servers: A Tutorial

Jack Y.B. Lee
*The Chinese University of Hong Kong*

**In conventional video-on-demand (VoD) systems, compressed digital video streams are stored in a video server for delivery to receiver stations over a communication network. This article introduces a framework for the design of parallel video server architectures and addresses three central architectural issues: video distribution architectures, server striping policies, and video delivery protocols.**

Among the many different types of media available for retrieval, retrieving full-motion, high-quality video in real time—video-on-demand (VoD)—poses the greatest challenges. Digital video not only requires significantly more storage space and transmission bandwidth than traditional data services, it must be delivered in time for continuous playback. Many studies conducted in the last decade have addressed these issues.

One common architecture shared by most VoD systems is a single-server model. The video servers can range from a standard PC for small-scale systems to massively parallel supercomputers with thousands of processors for large-scale systems. However, this single-server approach has its limitations.

## Scalability

The first of these limitations is capacity. When demand exceeds the server's capacity, one may need to replicate data to a new server. This doubles the system's storage requirements. To reduce storage overhead in replication and to balance the load among replicated servers, a number of studies have proposed replication algorithms based on video popularity as well as server heterogeneity, for example different storage and bandwidth.[1,2] A second approach partitions the video titles into disjointed subsets and stores each subset on a separate video server. Although this approach does not incur extra storage, it suffers from another problem—load balancing. Studies have shown that video retrieval is highly skewed in many applications because some videos are more popular than others.[3] Furthermore, the skewness changes with time, particularly when most users have seen a popular video and it becomes less popular. This implies that some video servers might become overloaded because of a popular title, while other servers might be underused.

A third approach takes advantage of the scale economy in large VoD systems by batching multiple users to share a single multicast video stream.[4] Multicasting popular video titles into multiple channels in a staggered manner means a user requesting a popular video title can be batched into one of these multicast channels and share it with other users. However, this approach does not support full VCR-like controls because the client can only select preset multicast channels. In a more recent study, Li and Liao proposed a way to merge users into a multicast stream that supports full VCR-like controls.[5]

The effectiveness of the batching approach depends on the video access pattern as well as the scale of the system. Loosely speaking, batching is most effective when a large portion of the users view only a small portion of video titles.

## Server fault tolerance

A second limitation of single-server VoD systems is that they cannot survive server failure. While replication can improve reliability, the storage requirement will multiply. Partitioning can confine a server failure to avoid bringing down all servers in the system, but it still suffers from the load-balancing problem discussed earlier. Finally, the batching approach is not intended to provide server fault tolerance and is also susceptible to server failure.

## Parallel video server

The scalability and fault tolerance problems are due to the fundamental limitations of the single-server architecture. Researchers have begun to investigate video server designs and implementations based on parallel video server architectures where video data are striped (rather than replicated or partitioned) across multiple servers. Designing video servers with parallel architectures not only breaks through the capacity limit of a single server, but also opens the way for server-level fault tolerance. Unlike replication and partition, server-level striping in parallel video servers achieves scalability without extra storage overhead. Moreover, the servers are load-balanced regardless of the skew in video popularities. This of course assumes each video stream is striped across all servers, which is likely because the stripe

1070-986X/98/$10.00 © 1998 IEEE

size is significantly smaller than the video stream.

This article introduces a framework for designing parallel video server architectures. I will address three central architectural issues: video distribution architectures, server striping policies, and video delivery protocols for parallel video servers.

## Parallel video distribution architectures

The essence of parallel video servers is the striping of data across an array of servers. Since the data receiver (such as a video decoder) expects a single stream of video data, data streams from each server must first be resequenced and merged. The term proxy refers to the system module responsible for resequencing and merging data from multiple servers into a coherent video stream for delivery to a client. In addition, the proxy can use data redundancy to mask server failures and achieve server-level fault tolerance.

The proxy is a software or hardware module that knows the system's configuration. This knowledge includes the number and addresses of servers, data locations, and striping policy. There are three ways to implement the proxy: at the server computer—proxy-at-server; at an independent computer—independent proxy; and at the client computer—proxy-at-client. The term computer in this context refers to the hardware performing the proxy function. In practice, this hardware may or may not be a computer in the general sense.

### Proxy-at-server

Figure 1 shows the proxy-at-server architecture, which includes $N_S$ server computers, each performing as both storage server and proxy. Because there are likely more clients than servers, each proxy will have to serve multiple clients simultaneously. The servers are connected locally by an interconnection network. The proxies combine data retrieved from local storage and other servers into a single video data stream for transmission to the clients.

Under this architecture, system configuration details can be completely hidden from clients. A drawback of this approach is processing and communications overhead. For example, to deliver $B$ bytes of video data from the video servers to a client, $B$ bytes of data must first be read from one or more servers' local storage. This data must then be transmitted via network to the client's proxy (unless the proxy happens to share the same host as the video server). Finally, the proxy processes the data and transmits to the client. Assuming all $N_S$ servers evenly service requests, then on the average $B(2N_S - 1)/N_S$ bytes of data transmission (server-to-proxy, proxy-to-client) and $B(2N_S - 1)/N_S$ bytes of data reception (proxy and client) are needed for every $B$ bytes of data delivered from the storage servers to a client.

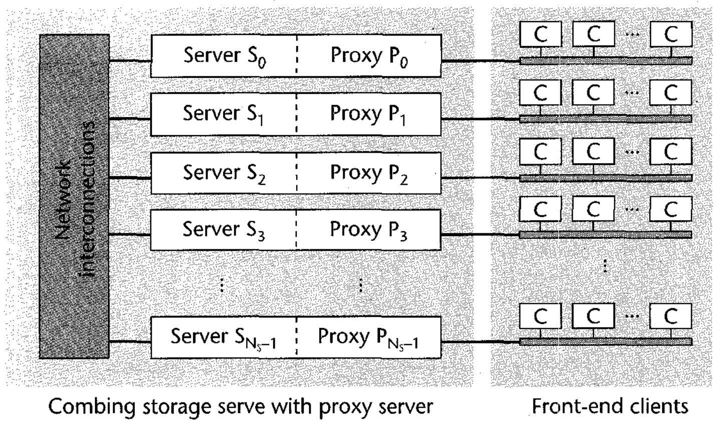
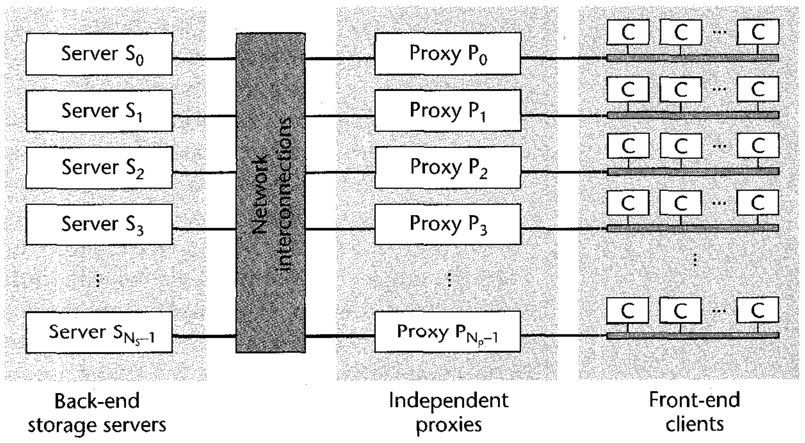


*Figure 1. The proxy-at-server architecture.*

### Independent proxy

Alternatively, separate computers can run the proxies. Figure 2 shows the independent proxy architecture using this approach. The back-end storage servers and proxy computers are connected locally by an interconnection network. Each proxy connects to multiple clients via another external network. Similar to the proxy-at-server architecture, this independent proxy architecture also hides server complexity from clients. Moreover, separating the proxy from the server eliminates interference between the two processes. This may simplify the server and proxy implementations.

*Figure 2. The independent proxy architecture.*



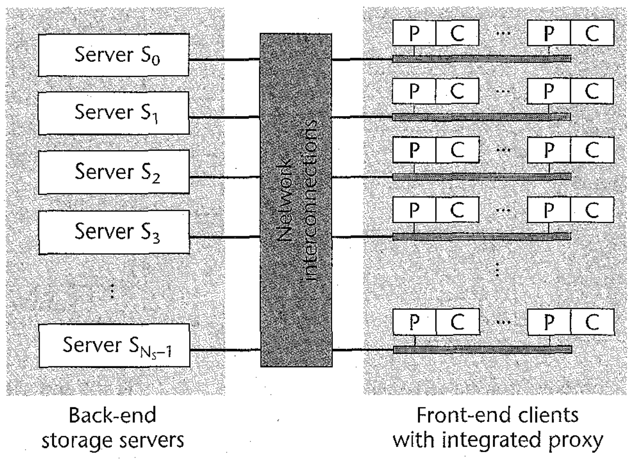


*Figure 3. The proxy-at-client architecture.*

Under the independent proxy architecture, data is first retrieved from the back-end server's local storage and then transmitted to the proxy. The receiving proxy then processes and transmits the data to clients. Therefore, requesting *B* bytes of data requires 2*B* transmission bytes and 2*B* reception bytes. This approach requires even more processing and network bandwidth than the proxy-at-server architecture. Moreover, additional hardware and network links are required to host and connect the proxies.

### Proxy-at-client

The third approach integrates the proxy into the client, as shown in Figure 3. This can be done by adding a proxy software module into the operating system or within the application. Under this architecture, a proxy requests the servers to send data directly to the client computer. After processing by the proxy, video data goes directly to the client application without further network communications. Hence, retrieving *B* bytes from the servers requires only *B* transmission bytes from the servers and *B* reception bytes at the client.

Compared with the proxy-at-server and independent proxy architectures, the proxy-at-client architecture requires only half the amount of data transfer and does not need separate hardware for the proxies. The primary advantage of the proxy-at-server and independent proxy architectures is client transparency (the ability of the proxy to hide the complexity of communicating with multiple servers). However, experimental results from a study by Lee and Wong have shown that the extra complexity involved is negligible, even while running the client in moderate-speed hardware such as a 60-MHz Pentium.[6]

On the other hand, if the computer running a proxy fails under the proxy-at-server and the independent proxy architecture, it will disrupt the service of all clients served by the proxy. Conversely, the same situation will only affect a single client in the proxy-at-client architecture because each proxy serves only one client and the proxy runs at the client host.

### Existing architectures

The proxy-at-server architecture has been considered by several researchers.[7-9] In Tewari's paper, they called this the flat architecture. The same studies also considered a two-tier architecture equivalent to the independent proxy architecture. They called the proxy a delivery node, which retrieves video data from back-end storage nodes and delivers a single video stream to the client. The independent-proxy architecture was also considered in another study by Buddhikot et al.[10] Unlike previous works, the authors implemented the proxy functionality in their custom asynchronous transfer mode (ATM) port interconnect controller (APIC), which also functions as the interconnection network linking the storage nodes and the external network. Lougher et al. considered using a striping server in a hierarchical network topology to perform the proxy functions.[11] Finally, the proxy-at-client architecture has been considered by several other researchers.[6,12-15]

All three architectures are scalable in the sense that more servers and proxies can be added to support more concurrent video sessions. However, the proxy-at-server and independent proxy architectures suffer from the problem that a proxy failure will affect all connected clients. Conversely, systems based on the proxy-at-client architecture do not have the proxy reliability problem; only back-end storage server failures need to be considered.

## Server striping policies

Striping is a general technique for distributing data over multiple devices to improve capacity (or throughput) and potentially reliability. Disk array and the Redundant Array of Inexpensive Disks (RAID)[16] are among the most successful applications. Other applications include network[17] and tape striping.[18] In a parallel video server, striping video data over multiple servers increases the system's capacity and potentially improves its reliability through data redundancy. This is called server striping. Striping a video stream across all $N_S$ servers is commonly called wide striping. Striping over a subset of the $N_S$ servers is called short striping.[8] Unless stated otherwise, wide striping is the method referred to in the following sections.

### Time striping

A video stream can be viewed as a series of video frames. Striping a video stream in units of frames across multiple servers is called time striping. Figure 4 depicts one example of how video units are striped using time striping. Assume that a stripe unit contains $L$ frames and the video plays at a constant frame rate of $F$ frames per second. In each round of $N_SL/F$ seconds, $L$ frames will be retrieved from each server and delivered to a client. In general, the striping size $L$ does not need to be an integer equal to or larger than one. In particular, if $L < 1$, then it is called subframe striping,[12] where $L \geq 1$ is simply frame striping.

Studies by Biersack et al. considered time striping by using a granularity of one frame and also one segment of a frame, or subframe striping.[12,19] For subframe striping, they divided a frame into $k$ equal-size units and distributed the units across the servers in a round-robin fashion. They argued that subframe striping has perfect load balancing for both constant bit-rate and variable bit-rate video streams, as each frame is striped equally across all servers. Conversely, a study by Buddhikot et al. used a stripe unit of $k$ ($k \geq 1$) frames.[10] They suggested solving the load balance problem by grouping more frames into a stripe unit to obtain a more uniform stripe unit size.

### Space striping

Time striping divides a video stream into fixed-length (in time) stripe units. A second approach would be to divide a video stream into fixed-size (in bytes) stripe units, called space striping. Space striping simplifies storage and buffer management at the servers because all stripe units are the same size. Moreover, the amount of data sent by each server in a service round is also the same. A video stream can be striped across the servers independent of the encoding formats and frame boundaries.

This space striping approach is employed by most of the studies already mentioned.[6-9,11,13-15] Depending on the system design, the stripe unit size can range from tens of kilobytes to hundreds of kilobytes. In most of the studies, a stripe unit is assumed to play back in a constant time length. However, under modern video compression algorithms like MPEG, a fixed-size stripe unit will likely contain a variable number of frames and partial frames. Moreover, if the video is compressed using constant-quality compression algorithms, then the video bit-rate will also vary. Consequently, the decoding time for a stripe unit at the client will vary and may cause playback starvation. To solve this problem, designers can incorporate the decoding time variation into the model to derive the client buffer required to compensate for the decoding time variations.

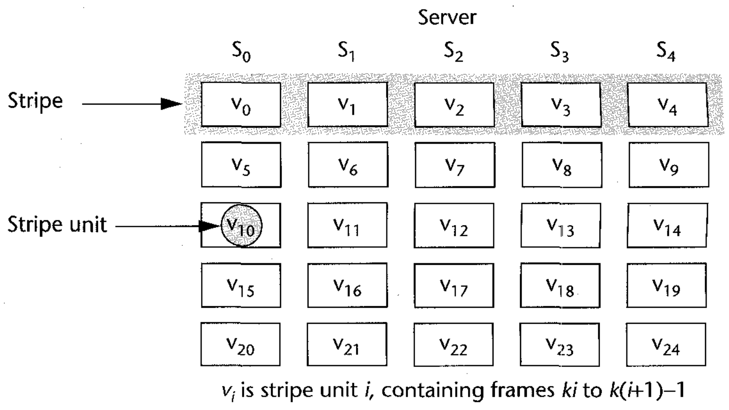

Server
$S_0$
$S_1$
$S_2$
$S_3$
$S_4$
Stripe
Stripe unit
$v_0$ $v_1$ $v_2$ $v_3$ $v_4$
$v_5$ $v_6$ $v_7$ $v_8$ $v_9$
$v_{10}$ $v_{11}$ $v_{12}$ $v_{13}$ $v_{14}$
$v_{15}$ $v_{16}$ $v_{17}$ $v_{18}$ $v_{19}$
$v_{20}$ $v_{21}$ $v_{22}$ $v_{23}$ $v_{24}$
$v_i$ is stripe unit $i$, containing frames $ki$ to $k(i+1)-1$

***Figure 4. Striping a video stream over five servers using time striping.***

### Placement policies

In the previous discussions, I have assumed a round-robin placement of the stripe units across the servers in the system. If the stripe units of a video stream are denoted using $v_0$, $v_1$, ..., and so on, then stripe unit $v_i$ will be stored in server ($i$ mod $N_S$). However, a minor problem with this policy is that server $i$ will likely store more stripe units than server $j$, for $i < j$. This happens because the video length is not always an integral multiple of the stripe size. Therefore, the last stripe will likely contain less than $N_S$ stripe units filling from server zero. To balance the storage, the round-robin policy can be modified to start striping a new video stream from different servers. Apart from round-robin placement, Tewari et al. also considered a random placement policy where the order within a stripe is permuted pseudo-randomly.[8] They suggested that the round-robin placement policy can introduce a convoy effect when one server becomes overloaded. That is, the overloading condition will shift from one server to the next due to the round-robin placement. This convoy effect can be avoided by permuting the order of the stripe units in each stripe.

### Issues in load balancing

In time striping, the frequency at which video frames are retrieved is the same for all participating servers. However, the size (in bytes) of stripe

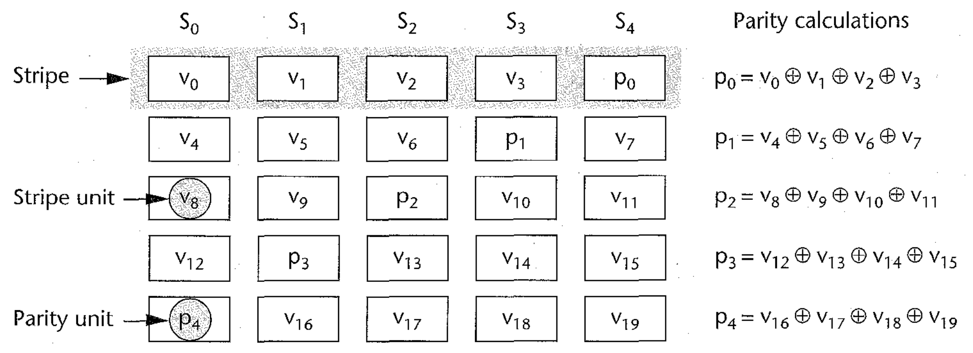

$S_0$
$S_1$
$S_2$
$S_3$
$S_4$
Parity calculations
Stripe
Stripe unit
Parity unit
$v_0$ $v_1$ $v_2$ $v_3$ $p_0$
$v_4$ $v_5$ $v_6$ $p_1$ $v_7$
$v_8$ $v_9$ $p_2$ $v_{10}$ $v_{11}$
$v_{12}$ $p_3$ $v_{13}$ $v_{14}$ $v_{15}$
$p_4$ $v_{16}$ $v_{17}$ $v_{18}$ $v_{19}$
$p_0 = v_0 \oplus v_1 \oplus v_2 \oplus v_3$
$p_1 = v_4 \oplus v_5 \oplus v_6 \oplus v_7$
$p_2 = v_8 \oplus v_9 \oplus v_{10} \oplus v_{11}$
$p_3 = v_{12} \oplus v_{13} \oplus v_{14} \oplus v_{15}$
$p_4 = v_{16} \oplus v_{17} \oplus v_{18} \oplus v_{19}$

*Figure 5. Adding redundant data to support server-level fault tolerance.*

units may not be the same. This is especially significant in video streams compressed using interframe compression algorithms like MPEG-1 and MPEG-2. Consequently, the amount of video data retrieved from each server in a round depends on the type of frames stored. Take MPEG-1 as an example. There are three frame types, namely I, P, and B, in order of decreasing average size. A server storing an I frame will store and send much more data than a server storing a B frame in the same round. Worse still, if the number of servers $N_S$ is a multiple of the MPEG group of pictures (GOP) size $G$, then the uneven load will be repeated for all future service rounds. This could lead to load imbalance where some servers become overloaded while others are underused.

For frame striping, this load-balancing problem can be reduced by selecting $L$ equal to integral multiples of the MPEG GOP size $G$. This can reduce the size differences between the stripe units. In practice, the size of each GOP differs slightly even in fixed-GOP MPEG encoding. Moreover, some encoders produce MPEG streams having variable GOP sizes to improve visual quality, thereby defeating the discussed solution.

Conversely, Biersack et al. proposed subframe striping to achieve perfect load balancing.[12,19] However, subframe striping suffers from the problem that the processing complexity at the proxy will increase when more servers are added to the system. This is because the proxy needs to combine subframe data from all servers for each video frame. On the other hand, space striping is balanced by definition. To cope with variable playback duration for each stripe unit, additional client buffering can be used.

Another problem arises when a video stream is played at a different speed than normal playback speed, like fast forward and fast backward. Two approaches support fast forward: encode separate streams for fast forward and fast backward purposes; and skip frames to achieve an apparent faster playback rate. The first approach needs extra storage space, but the system design and implementation are fairly straightforward. For the second approach, the fast forward feature leads to load-balancing problems for parallel video servers. To see why, consider a system having four servers and using time striping of one frame per stripe unit. If double-speed fast forward is implemented

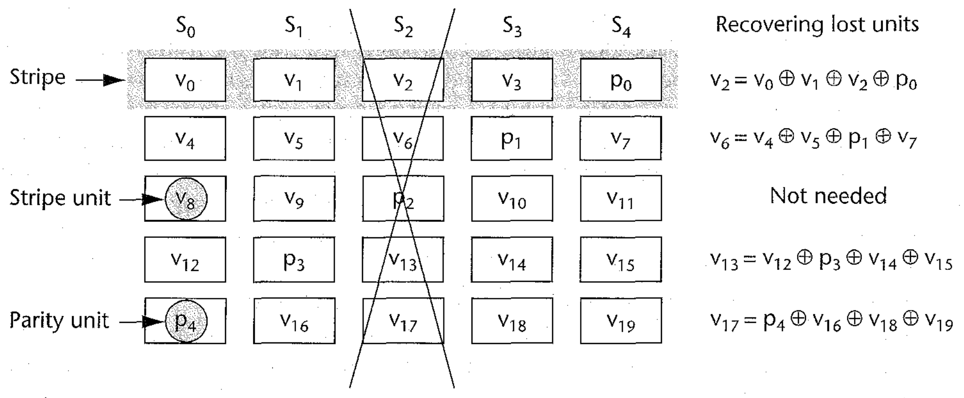

$S_0$
$S_1$
$S_2$
$S_3$
$S_4$
Recovering lost units
Stripe
Stripe unit
Parity unit
$v_0$ $v_1$ $v_2$ $v_3$ $p_0$
$v_4$ $v_5$ $v_6$ $p_1$ $v_7$
$v_8$ $v_9$ $p_2$ $v_{10}$ $v_{11}$
$v_{12}$ $p_3$ $v_{13}$ $v_{14}$ $v_{15}$
$p_4$ $v_{16}$ $v_{17}$ $v_{18}$ $v_{19}$
$v_2 = v_0 \oplus v_1 \oplus v_2 \oplus p_0$
$v_6 = v_4 \oplus v_5 \oplus p_1 \oplus v_7$
Not needed
$v_{13} = v_{12} \oplus p_3 \oplus v_{14} \oplus v_{15}$
$v_{17} = p_4 \oplus v_{16} \oplus v_{18} \oplus v_{19}$

*Figure 6. Recovering stripe units lost due to failure of server 2.*

simply by skipping every other frame in the video stream, then two of the four servers will handle all data transmissions while the other two sit idle. This problem has been studied in detail by Wu et al and Buddhikot et al.[9,10] They proposed algorithms for the data layout, scheduling, and playout control to support fast forward and other interactive features.

### Issues in redundancy

As discussed earlier, parallel video servers open the way to achieving fault tolerance at the server level. Ideally, the system should be able to maintain continuous video playback for all active sessions when one or more of the servers become inoperable. As with the disk array and RAID architectures, data redundancy can be added to support failure recovery in a parallel video server. The basic idea is to introduce one or more parity units into each stripe. As shown in Figure 5, the parity units are computed using the rest of the stripe units in the same stripe. When a server fails, the lost stripe unit stored in the failed server can be recovered from the parity unit together with the remaining stripe units (Figure 6).

For single-failure protection, simple parity computed from exclusive-or between the data stripe units can be used. A higher level of redundancy can be achieved by using more sophisticated erasure-correction codes such as the Reed-Solomon code. Note that to perform erasure-correction, stripe units within a stripe must be of identical size. This requirement argues against time-striping algorithms, which result in variable stripe unit sizes. Conversely, space striping, by definition, has fixed stripe size and can easily extend to incorporate redundancy.

For parallel video servers, the fault tolerance issue has been studied by Wong et al. and Biersack et al.[6,12] Biersack's system employs subframe striping, hence they can use error-concealment techniques to partially mask a server failure. Alternatively, they also suggested the possibility of using forward error correction (FEC) to recover subframes lost in a failed server. The study by Wong et al. employs space striping algorithms similar to RAID to achieve server-level fault tolerance.[6] Their system can sustain nonstop video playback for all clients when a server fails.

In another study, Bolosky et al. proposed using mirroring to achieve server fault tolerance.[13] In particular, they used declustering to evenly distribute the extra load caused by a server failure to the remaining active servers. Unlike the study by Wong et al., their system does not perform real-time recovery when a server fails; it cannot sustain nonstop video playback for existing users.

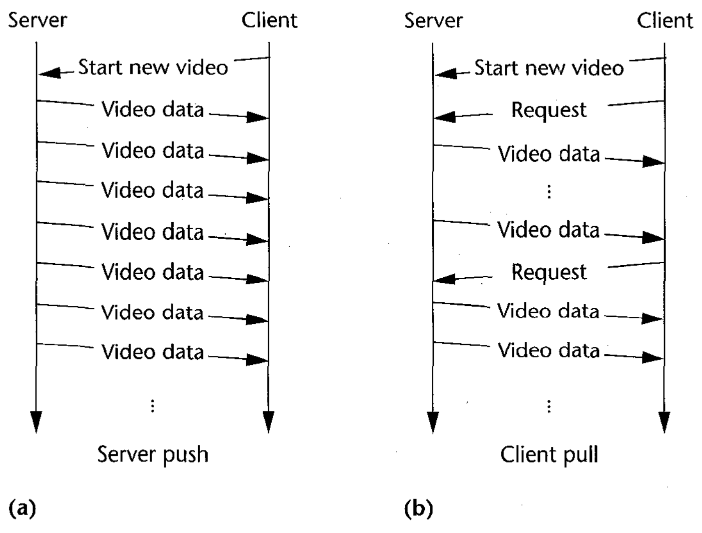


*Figure 7. Service model for VoD systems: (a) server-push; (b) client-pull.*

## Parallel video delivery protocols

The parallel video delivery requirement poses challenges in designing the application protocol's flow control, error control, and synchronization. The following section focuses on the client-pull versus the server-push service model, then discusses synchronization and fault-tolerance issues.

### Client-pull versus server-push

VoD systems generally have two ways to request and deliver video data from a server to a client. Most VoD systems let the video server send data to the client at a controlled rate. The client receives and buffers the incoming video data for playback through a video decoder. As shown in Figure 7a, once the video session starts, the video server continues the data transmissions until the client specifically sends a request to stop it. Since the server pushes video data to the client at a controlled rate, this approach is called server-push.

In the traditional request-response model, the video client sends a request to the server for a particular piece of video data. As shown in Figure 7b, upon receiving the request the server retrieves the data from the disk and sends it back to the client. This approach is called client-pull for obvious reasons. The studies by Lee and Wong employ this client-pull service model.[6,15]

### Interserver synchronization

Most studies on single-server VoD systems employ the server-push delivery model. This

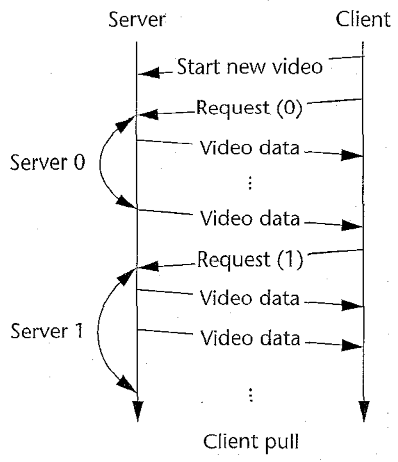


*Figure 8. Extended client-pull service model for use in a parallel video server.*

model allows the system designer to devise periodic schedules at the server for reading data off the disk and then transmitting it to the client. Extending the server-push model to a parallel video server causes a new problem due to the parallel transmissions from multiple independently running servers.

For example, consider a system with $N_S$ servers using fixed-size space striping. To start a new video session, a client sends a request to the proxy, which in turn sends requests to all $N_S$ servers to start a new video session. Due to delay variations in processing, networking, and scheduling, the servers will start transmitting data at different times. The first stripe unit might even arrive at the proxy later than the subsequent stripe units. Consequently, the proxy has to buffer the later stripe units to wait for the first stripe unit to arrive for playback. This increases the client buffer requirement and startup delay.

This synchronization problem has been studied by Biersack et al.[19] For scenarios where network delays between servers and a client is different, they propose adding different delays to the starting times of each server to compensate for delay differences. They also extended this model to include bounded network delay jitters. In another study by Buddhikot et al., they designed a closely coupled system in which each storage node is connected by a custom high-speed interconnection network (APIC).[10] The proximity of the storage nodes enables them to be accurately synchronized through the common APIC. Lee et al. extended the client-pull service model for use in a parallel video server (Figure 8).[15] Since a client explicitly sends a request to a particular server for a specific piece of video data, the synchronization is implicit and the servers need not be separately synchronized.

## Detecting and masking server failures

Earlier, I discussed how data redundancy can be introduced among the servers to support server-level fault tolerance. The redundant data allows the receiver to mask a server failure by computing lost stripe units stored in the failed server from the parity units together with the remaining stripe units. In this section, I focus on ways to deliver redundant data to the receivers.

**Forward error correction.** Network communications, generally offer two ways to recover packets lost in transit. The first way, called forward error correction (FEC), sends redundant data along with normal data to the receiver at all times. In this way, the receiver can recover lost packets by using the received data together with the redundant data. This approach can extend to parallel video servers to recover lost stripe units due to server failures.[6,12]

FEC has the distinct advantage that the receiver does not need to detect a server failure. Because the receiver always receives redundant data, lost stripe units can readily be recovered in case a server fails. However, like the case in network communications, FEC incurs constant transmission overhead even when no server fails. According to coding theory, one redundant symbol is needed for every lost symbol. Therefore, by using $K$ to denote the number of lost symbols we want to recover per parity group (or stripe), the transmission overhead will be $K/(N_S - K)$. This overhead could become significant for systems having a small number of servers or a high level of redundancies.

For a redundancy level of more than one (for example, $K > 1$), Wong et al. proposed a progressive redundancy transmission algorithm that transmits less redundant symbols at startup.[6] A server failure prompts a request for the servers to start transmitting one more redundant symbol per stripe. For example, let $K = 3$. Then the system can transmit only one redundant symbol at startup. When a server fails, the system will be able to mask the failure immediately using the available redundant symbol. At the same time, the receiver will request the servers to start transmitting one more redundant symbol per stripe, and so on. As servers seldom fail simultaneously, this algorithm can keep the transmission overhead low while still allowing the system to survive multiple server failures.

**On-demand correction.** A second way to recover lost packets in network communications is retransmission. Unlike FEC, retransmission requests extra transmission only when needed.

*Table 1. Summary of design choices studied by various researchers.*

| Researchers | Video Distribution Architecture | Server Striping Policy | Video Delivery Protocol | Server Fault Tolerance |
|---|---|---|---|---|
| Biersack et al. (Video server array) | Proxy-at-client | Time | Server path | Striping w/parity; FEC |
| Bolosky et al. (Tiger video fileserver) | Proxy-at-client | Space | Server path | Mirroring with declustering |
| Buddihikot et al. (MARS) | Independent proxy | Time | Server path | — |
| Freeman et al. (SPIFFI) | Proxy-at-client | Space | — | — |
| Lee et al. (Server array & RAIS) | Proxy-at-client | Space | Client pull | Striping w/parity; FEC and ODC |
| Lougher et al. | Independent proxy | Space | — | — |
| Reddy et al. | Proxy-at-client, Independent proxy | Space | Server push | — |
| Tewari et al. (Clustered video server) | Proxy-at-client, Independent proxy | Space | Server push | — |
| Wu and Shu | Proxy-at-client, Independent proxy | Space & Time | Server push | — |

While retransmission cannot recover packets lost in failed servers, this on-demand principle can be extended for the delivery of redundant data. This approach is known as on-demand correction (ODC). Specifically, the system does not send redundant data until the system detects a server failure. This method completely avoids the transmission overhead in FEC and the system can still recover from server failures.

The challenge in ODC is to devise a way to detect server failures quickly and reliably. The detection method must be quick enough to ensure that video playback continuity can be sustained while the system requests redundant data for recovery. On the other hand, the detection method must not generate too many false alarms or risk causing the system to send redundant data anyway.

Wong et al. designed a new video transfer protocol together with a new network protocol to detect and mask server failure.[6] They successfully implemented both FEC and ODC in a LAN environment where network delay is relatively short and constant.

## Future research directions

Many of the studies reviewed here have shown that a scalable video server can be built from a cluster of cheaper, less powerful servers. Table 1 summarizes the design choices studied by various researchers. In addition, some of the studies have exploited parallelism to support server-level fault tolerance. The early results have demonstrated the feasibility of maintaining nonstop video services despite server failures.

Despite its brief history, parallel video server research has obtained many encouraging results. As discussed in this article, many architectural and design alternatives with many combinations remain unexplored. In the future, the following issues will need study:

1. Server synchronization issues for loosely-coupled, push-based parallel video servers. The synchronization scheme must be efficient, yet accurate enough to avoid server asynchrony. Furthermore, the scheme must be able to survive server failures.

2. Push-based transmission scheduling for loosely-coupled parallel video servers. The scheduling policy must tolerate the clock jitter inherent among loosely coupled servers. Moreover, the scheduling policy must handle the case when one or more server fails and be able to sustain nonstop video playback for existing users.

3. Scheduling issues for delivering VBR video streams in a parallel video server.

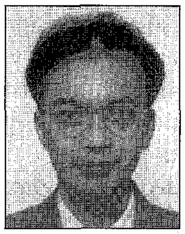
4. Extensions of the server striping principle to implement parallel multimedia servers. **MM**

## Acknowledgments

I thank the reviewers for their constructive comments in improving this article to its final form.

## References

1. N. Venkatasubramanian and S. Ramanthan, "Load Management in Distributed Video Servers," *Proc. 17th Int'l Conf. on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 528-535.
2. C.C. Bisdikian and B.V. Patel, "Issues on Movie Allocation in Distributed Video-on-Demand Systems," *Proc. ICC95*, IEEE Press, Piscataway, N.J., 1995, pp. 250-255.
3. 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," *Proc. Multimedia 97*, ACM Press, New York, 1997, pp. 349-357.
4. A. Dan, D. Sitaram, and P. Shahabuddin, "Scheduling Policies for an On-Demand Video Server with Batching," *Proc. 2nd ACM Multimedia Conf.*, ACM Press, New York, 1994, pp. 15-24.
5. W. Liao and V.O.K. Li, "The Split and Merge Protocol for Interactive Video-on-Demand," *IEEE MultiMedia*, Vol. 4, No. 4, Oct. 1997, pp. 51-62.
6. P.C. Wong and Y.B. Lee, "Redundant Array of Inexpensive Servers (RAIS) for On-Demand Multimedia Services," *Proc. ICC 97*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 787-792.
7. A. Reddy, "Scheduling and Data Distribution in a Multiprocessor Video Server," *Proc. Second IEEE Int'l Conf. on Multimedia Computing and Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 256-263.
8. R. Tewari, R. Mukherjee, and D.M. Dias, "Real-Time Issues for Clustered Multimedia Servers," IBM Research Report RC20020, June 1995.
9. M. Wu and W. Shu, "Scheduling for Large-Scale Parallel Video Servers," *Proc. Sixth Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 126-133.
10. M.M. Buddhikot and G.M. Parulkar, "Efficient Data Layout, Scheduling and Playout Control in MARS," *Proc. NOSSDAV 95*, Springer-Verlag, Berlin, 1995, pp.318-329.
11. P. Lougher, D. Pegler, and D. Shepherd, "Scalable Storage Servers for Digital Audio and Video," *Proc. IEE Int'l Conf. on Storage and Recording Systems 1994*, IEE Press, London 1994, pp. 140-3.
12. C. Bernhardt and E. Biersack, "The Server Array: A Scalable Video Server Architecture," *High-Speed Networks for Multimedia Applications*, Kluwer Press, Boston, 1996.
13. W.J. Bolosky et al., "The Tiger Video Fileserver," *Proc. Sixth Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, IEEE Computer Society Press, Los Alamitos, Calif., 1996.
14. C.S. Freedman and D.J. DeWitt, "The SPIFFI Scalable Video-on-Demand System," *Proc. ACM Sigmod 95*, ACM Press, New York, June 1995, pp. 352-363.
15. Y.B. Lee and P.C. Wong, "A Server Array Approach for Video-on-Demand Service on Local Area Networks," *IEEE Infocom 96*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 27-34.
16. D.A. Patterson et al., "Introduction to Redundant Array of Inexpensive Disks (RAID)," *Compcon Spring 89*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 112-17.
17. C. Brendan, S. Traw, and J.M. Smith, "Striping Within the Network Subsystem," *IEEE Network*, IEEE Press, Piscataway, N.J., 1995, pp. 22-32.
18. A.L. Drapeau and R.H. Katz, "Striped Tape Arrays," *Proc. 12th IEEE Symp. on Mass Storage Systems*, IEEE Press, Piscataway, N.J., 1993, pp. 257-65.
19. E. Biersack, W. Geyer, and C. Bernhardt, "Intra- and Inter-Stream Synchronization for Stored Multimedia Streams," *Proc. IEEE Int'l Conf. on Multimedia Computing Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 372-381.



**Jack Lee** is a visiting assistant professor in the Information Engineering Department at the Chinese University of Hong Kong. He received his PhD and Beng degrees from the same department in 1997 and 1993. His research interests include distributed multimedia systems, fault-tolerant systems, and Internet computing.

Readers may contact Lee at the Department of Information Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong, email yblee@ie.cuhk.edu.hk.