A SERVER-LESS ARCHITECTURE FOR BUILDING SCALABLE, RELIABLE, AND COST-EFFECTIVE VIDEO-ON-DEMAND SYSTEMS

LEUNG WAI TAK

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
In
Information Engineering

July 2002


香港中文大學
圖書館系統
28 MAR 2003
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

ACKNOWLEDGEMENT

First of all, I would like to thank my supervisor, Prof. Jack Y. B. Lee, for his invaluable advices and guidance. During my research study, he fully supported my work and provided guidance on my research direction. I would also like to thank Professor John C. S. Lui and Professor Raymond W. H. Yeung for their suggestions in modeling the system reliability in section 5.1.

I am grateful to my friends, Hing, Lun, and Rudolf, for their kind discussions and helps during my study. I also appreciate the helps and funs all my friends gave me over these two years.

Finally, I would like to express my gratitude to my family (especially my mom) and RonEmily for their support and love throughout my life.

ABSTRACT

Video-on-demand (VoD) systems have traditionally been built around the client-server architecture, where a video server stores compressed video for delivery to clients connected by a network. With increasing demand for large-scale VoD systems, researchers have spent considerable effort in designing scalable, reliable, and cost-effective video servers. Nevertheless, a video server can only have finite capacity. As the system scales up, the server will need to be upgraded and this becomes increasingly expensive as the system scales beyond thousands of users. In this thesis, we investigate a radically different architecture where the bottleneck – video server, is eliminated altogether. Specifically, this peer-to-peer, server-less architecture relies on the client machines for distributed data storage and delivery. A client initiating a new streaming session will first locate other clients where the requested stream is stored, and then requests delivery of the stream directly from those clients instead of from a central server. This fully distributed architecture is inherently scalable as the storage and delivery capacity grows with the number of clients in the system. Additionally, we develop fault-tolerance algorithms for the system so that stream delivery can be maintained even if some of the clients fail. Our results show that the system can be scaled up to over 1000 nodes with reliability even exceeding those of dedicated video servers.

摘要

傳統的視頻點播 (VoD) 系統是採用伺服器對用戶端 (client-server) 的結構，當中伺服器將視像資料透過網路傳送到用戶端播放。在對大規模的視頻點播系統的需求不斷增加下，已有不少的研究去設計高升級性的、可靠的、及高成本效益的視頻伺服器。但是伺服器的容量始終是有限的。當系統的規模不斷擴大，伺服器便需要升級來增加容量。特別是當用戶數目達至數以千計時，伺服器升級所需的成本更是十分昂貴。在本論文中，我們研究一種突破性的結構，將傳統系統中的容量限制——伺服器移除。更具體的說，這種端點對端點 (peer-to-peer)、無伺服器 (server-less) 的結構是使用用戶端的電腦作爲分散式 (distributed) 視像資料的儲存及傳送。當用戶端要求播放視像時，用戶端首先需要找出系統中其他儲存視像的用戶端，並且要求其他用戶端將視像資料傳送過來播放，而並非對伺服器提出要求。由於系統的儲存及傳送容量會隨着用戶端數目的增長而提升，所以這分散式結構擁有十分高的可升級性。除此之外，透過我們提出的容錯 (fault-tolerance) 機制，即使部分用戶端離線，這系統仍可維持視像資料的傳送。我們得出的數值結果顯示這系統的規模可擴大至超過一千個用戶端，並且系統的可靠性更可超越專業的視頻伺服器。

Contents

Chapter 1

INTRODUCTION

Current video-on-demand (VoD) systems are commonly designed around the client-server architecture. Under this architecture, a client sends a request to a video server for a video title and then the server transmits video data to the client for playback. As the number of user increases, the server will eventually reach its capacity limit. To further increase the system capacity, one can add more servers and distribute the requests to them, such as distributed server [1-2] and parallel server [3-7] architectures. As the system load is shared among multiple servers, the total system capacity can then be extended when more servers are added to the system.

Nevertheless, the cost of upgrading servers can be substantial, as video servers typically require high-end server hardware with high I/O bandwidth, large memory capacity, as well as storage capacity. Even in the best case, such as parallel server and distributed server architectures that do not require data replication, the server cost will still increase at least linearly with the traffic demand. Moreover, apart from server cost, the distribution network will also need to be upgraded with more bandwidth to carry the vast amount of video traffic to the users. Given the high cost of long-distance backbone networks, it is no wonder why metropolitan-scale VoD services are still uncommon in practice.

In this study, we take a radically different approach to building large-scale, scalable, reliable, and cost-effective VoD systems. In particular, we turn our attention to an often neglected element in a VoD system – the client-side device or commonly called the set-top box (STB).

Developments of STB have continued for many years and current STBs not only are low cost, but also are relatively powerful due to the rapid technological development and the economy of scale achieved by the personal computer industry. While early generations of STB are very limited in function and capability, the current trend in STB development is towards evolving from a simple video-receiving and decoding device into a home entertainment center with functions like VoD, TV-over-Internet, harddisk-based personal video recorder, messaging center, web browser, CD player, DVD player, digital audio jukebox, or even game console. This evolution not only greatly enhances the usefulness of a STB, but also opens a radically new way to build VoD systems.

This motivates us to take advantage of the increased storage and processing capability of STBs to build a completely distributed VoD system that does not require dedicated server at all. We call this a server-less architecture for obvious reason. In this server-less architecture, all STBs, or called nodes in this thesis, in the system serve both as a client and as a mini-server. Video data are distributed among the nodes and multiple nodes work together to serve video streaming requests from other nodes. The beauty of this architecture is that the system is inherently scalable, i.e., when new users are added to the system, they add both streaming load and streaming capacity to the system. Moreover, network costs can also be reduced because the nodes are likely to be clustered together, reducing the need for costly long-distance network backbone.

However, building a server-less VoD system is not without challenges. First, the nature of a distributed architecture calls for a data distribution and placement policy. Even more challenging is the fact that the system not only needs to correctly deliver data from distributed hosts, but also needs to deliver them in time to ensure continuous, jitter-free video playback. Second, in a distributed system with potentially thousands or more autonomous nodes, node failures will become the norm rather than the exception. Therefore a fault tolerant mechanism to sustain node failures becomes an essential part of the system. Third, as nodes join and leave the system, we need to redistribute the data and load to ensure load balance (nodes joining) and to maintain system reliability (nodes leaving). Fourth, in practice nodes in the same cluster themselves may not be homogenous in terms of capability (e.g. different storage capacity, network bandwidth) and/or load (e.g. multi-user nodes). Finally, to address the needs of commercial service providers, one will also need a content protection mechanism to prevent ordinary users from pirating the video contents in a bit-for-bit manner.

Obviously there are many more challenges in addition to the above five for such a radically different architecture. Our goal in this study is to establish the feasibility of the server-less architecture and to investigate its basic properties such as resource requirement and system performance. So we address the first two of the above-mentioned challenges in this study. The remaining challenges involve further extending the architecture to adapt to dynamic reconfiguration, heterogeneous nodes and content protection, so they are left for future work.

In this thesis, we show the feasibility of building VoD systems based on the server-less architecture. We present a complete system design, including a data placement policy, a retrieval and transmission scheduler, and a fault tolerant

mechanism, and analyze the system's resource requirements and performance. Our results show that it is feasible to build a VoD system using the server-less architecture with resources available in current STBs or ordinary PCs, such as buffer and storage capacity. Moreover, the system can achieve system-level reliability using the proposed fault tolerance mechanism, comparable to or even exceeding those of dedicated high-end video servers.

The rest of the thesis is organized as follows: Chapter 2 reviews related works on large-scale video-on-demand systems and large-scale distributed file systems; Chapter 3 presents the proposed server-less VoD architecture; Chapter 4 presents a performance model for the architecture; Chapter 5 studies system reliability issues; Chapter 6 evaluates performance of the architecture using numerical results; Chapter 7 investigates a multiple parity groups architecture to enhance scalability; and Chapter 8 concludes the thesis.

Chapter 2

RELATED WORKS

In this chapter, we first review in section 2.1 a number of previous related works and then compare them with our approach in section 2.2.

2.1 Previous Works

Current VoD architectures can be classified into centralized [8] and distributed [1-7] [9-10] architectures. In centralized architectures, only the central server serves user requests and so it becomes the system's primary bottleneck assuming the network bandwidth is sufficient. By contrast, requests are shared by multiple servers in distributed server architectures such that capacity can be scaled up to beyond that of a single server by adding more servers to the system.

Serpanos, *et al.* [8] compared the performance of centralized and distributed architectures for video servers. They concluded that in general a centralized architecture is preferable in terms of performance and management, but at the expense of higher cost. To improve cost effectiveness, distributed [1-2] or parallel [3-7] server architectures are commonly employed. For example, one can replicate video data to multiple servers to share the load. To reduce the storage overhead incurred in video replication, one can limit replication to the more popular video titles. For example, On, *et al.* [1] studied replication assignment and update frequency in relation to the

desired data availability, consistency, and QoS requirement. Serpanos, *et al.* [2] proposed a MMPacking video assignment algorithm based on video popularity to achieve load and storage balance.

Another approach is the use of parallel server architectures studied by Lee [3-4], Bernhardt, *et al.* [5], Wu and Shu [6], and Tewari, *et al.* [9] that employ server-level data striping. Compared to replication and caching, parallel server architectures eliminate the need for data replication and are inherently load balanced. Moreover, one can introduce data and hardware redundancies into the system to achieve server-level fault tolerance [7] [9-10], making the system even more reliable than central-server systems.

Another area related to our study is the peer-to-peer (P2P) architecture popularized by software systems such as Napster [11] and Gnutella [12]. These P2P systems are primary designed to function as a large distributed storage system [13-14]. In a P2P system, a user shares data with a group of other users. A user can search for the desired data by submitting a query to its neighbors or to a directory server. Once the desired data are located, the user downloads the data directly from the other user's computer. As the data are selectively replicated among user nodes, this structure allows sharing of data by a large community at low cost, as a dedicated server is no longer needed. The main challenge comes from the complexity in distributing replicated data to achieve load balance and fault tolerance [14]. As hosts in a P2P system have varying network bandwidth and processing capability, quality-of-service cannot be guaranteed, and slow or even broken connections are not uncommon. Nevertheless, the ease of setting up and participating in a P2P system and the need for a decentralized data-sharing platform has outweighed these limitations.

Other distributed file systems such as Coda [15], Farsite [16] and OceanStore [17], have also been proposed and studied over the years. Coda is developed by Satyanarayannan *et al.* [15] to provide a highly available file system for distributed workstation environment. It employs the client-server architecture and uses server-level replication to achieve high availability with tradeoffs in data consistency. A client connects to the server upon user requests and caches locally the needed data to provide continued access during temporary failures such as network failure. Farsite is developed by Bolosky, *et al.* at Microsoft as a distributed file system [16]. Farsite does not assume mutual trust among the client machines and their research focus is to provide security, availability, and reliability by distributing multiple encrypted replicas among the client machines. Bolosky, *et al.* concluded that Farsite is feasible on the current desktop infrastructure in the commercial environment based on measurements of file-system space, machine availability, and machine load of the client machines at Microsoft. Finally, Kubiatowicz *et al.* [17] developed OceanStore as a distributed storage system built from untrusted machines to achieve global-scale persistent storage. Data in OceanStore are protected through redundancy and cryptographic techniques. Data replicas are free to migrate among geographically distributed servers so as to achieve global file availability in case of regional failures.

2.2 Contributions of this Study

Compared to traditional client-server architectures, our approach decentralizes and distributes the server functions to the clients. This server-less architecture eliminates the primary bottleneck – video server, from the system. This not only improves the scalability of the system, but also eliminates the costs of the expensive video servers. Moreover, despite the fact that nodes in a server-less VoD system have significantly

lower reliability, we can achieve system-level reliability exceeding even that of dedicated video servers by means of erasure-correcting codes and fault tolerant video streaming techniques.

Compared to current P2P systems such as Naptser and Gnutella, and distributed file systems such as Coda, Farsite and OceanStore, the architecture investigated in this study serves completely different applications, i.e. video-on-demand versus file sharing. Compared to file sharing, in addition to the challenge of data availability and content search/retrieval, a VoD system must be capable of guaranteeing stringent performance requirements that are essential to the correct operation of the system. The server-less VoD architecture investigated in this study is specifically designed to address these challenges for building VoD services comparable to or even better than existing client-server VoD systems.

This study, to the best of our knowledge, provides the first result to clearly demonstrate the feasibility of building VoD systems based on the server-less architecture. We present a complete system design, including a data placement policy, a retrieval and transmission schedulers, and a fault tolerant mechanism, and analyze the system's resource requirements and performance. The results show that it is indeed feasible to build VoD systems without video servers using today's computing and networking hardware, and more significantly, achieve comparable or even better performance and reliability than traditional VoD architectures with dedicated video servers.

Chapter 3

ARCHITECTURE

In this chapter, we present the details of the server-less architecture, including the data placement policy, the retrieval and transmission schedulers, and the fault tolerant mechanism. A server-less VoD system comprises a pool of user nodes connected by a network as shown in Figure 3.1. Each node has its own CPU, memory, and disk storage. Inside each node there is a mini video server software that serves a portion of each video title to other nodes in the system. Unlike conventional video server, this mini server streams a much lower aggregate bandwidth and therefore can readily be implemented in today's STBs and PCs. For large systems, the nodes can be further divided into clusters where each cluster forms an autonomous system that is independent from other clusters. The system designer can take advantage of clustering to group nodes according to their geographical locations or network topology to localize network traffic. Many other potential applications and optimizations are also possible and further research is warranted to investigate the challenges and opportunities of clustering in a server-less VoD system. In the rest of this study, we will focus on a single, independent cluster.

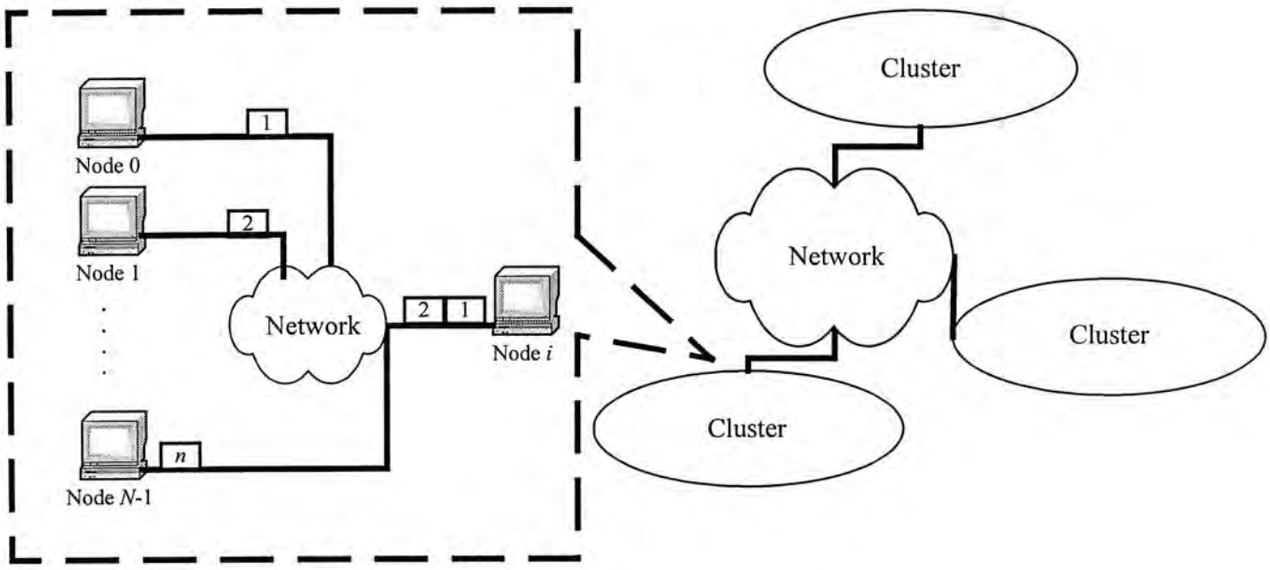

Figure 3.1. Architecture of a server-less VoD system.

3.1 Data Placement Policy

As discussed in section 2.1, distributed systems often employ data replication and caching to improve scalability. However, unlike video servers where storage capacity is usually large, a node in the form of a STB or a PC will have relatively limited storage capacity. Therefore we propose the use of data striping instead of replication to reduce the storage requirement. Another advantage of striping over replication is the amount of redundancy needed to achieve the same level of reliability. In particular, Weatherspoon *et al.* [18] concluded that systems based on striping and erasure-correcting codes use an order of magnitude less bandwidth and storage to provide similar system reliability as replicated systems.

Under the striping-based data placement policy, a video title is first divided into fixed-size striping units (or called blocks) of Q bytes each. And we will discuss in chapter 6.4 about the system parameters for determining the striping size Q. These

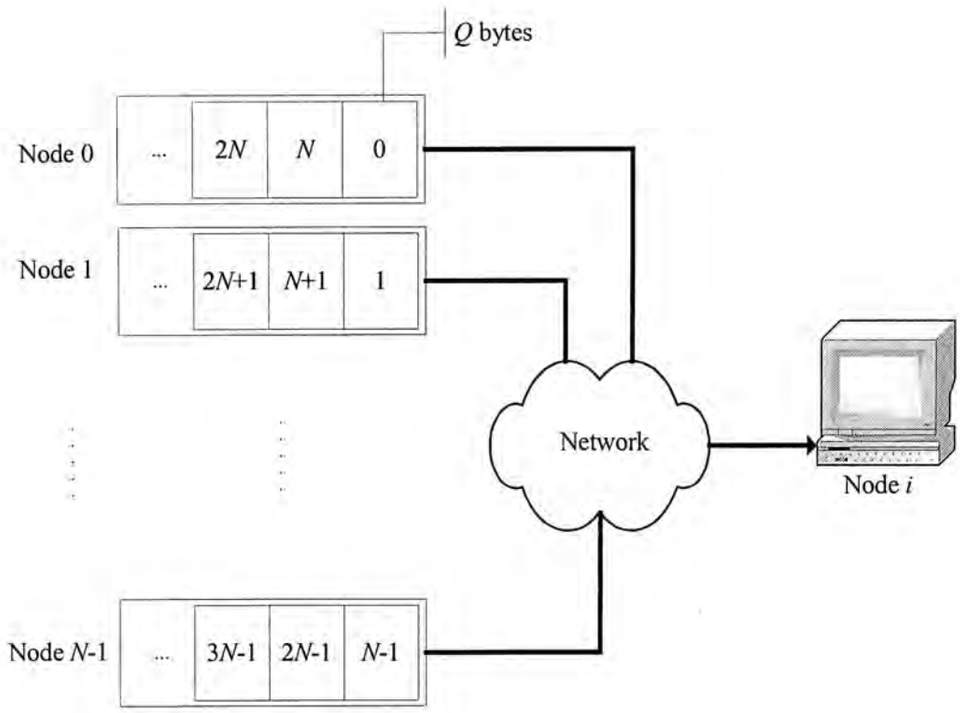

Figure 3.2. A striping-based data placement policy.

blocks are then distributed to all nodes in the cluster in a round-robin manner as shown in Figure 3.2.

Strictly speaking, small storage imbalance can still exist if one always starts placing the first block of a video title to the same node (e.g. node 0). This is because the total number of blocks for a video title may not be exact multiples of the number of nodes in the system and therefore, the last striping group may have fewer blocks than the number of nodes in the system. This leads to a small storage requirement difference of one stripe unit among nodes in the cluster as shown in the example in Figure 3.3.

To tackle this storage imbalance problem, we can decrease the block size of the last stripe of a video to evenly distribute data to all the nodes in the cluster. Let N be the number of nodes in the cluster. Suppose there are L ($L<N$) blocks in the last stripe of a video title, then we simply reduce the block size of the last stripe from Q to

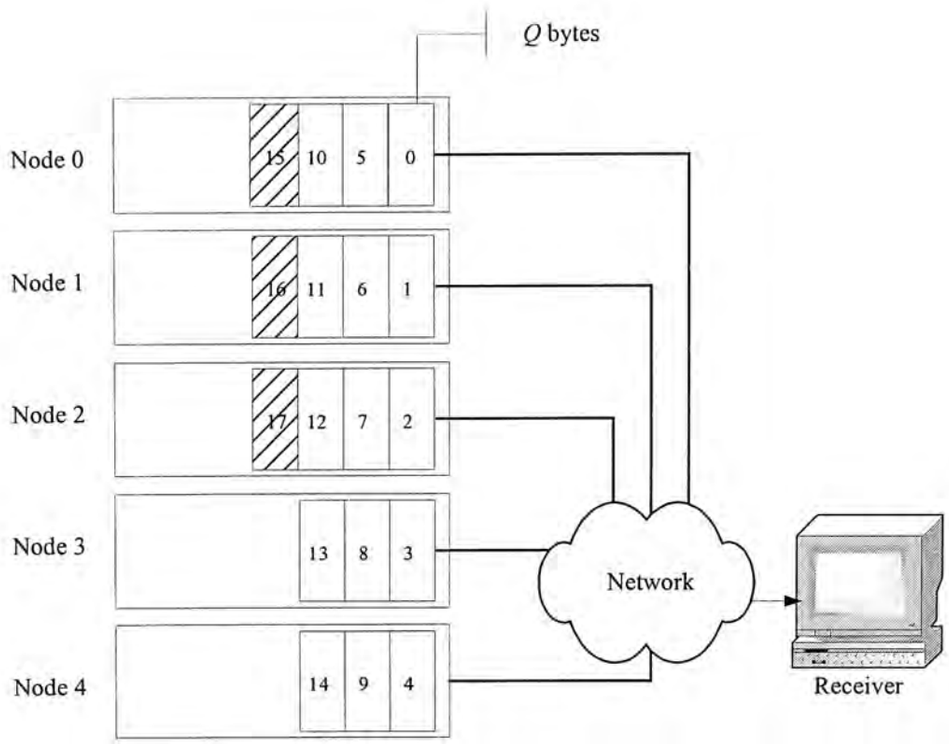

Figure 3.3. An example of storage imbalance of 5 nodes with 18 striping blocks.

$$Q' = \frac{L}{N} Q \tag{3.1}$$

bytes so that the remaining data will be evenly distributed to all nodes as shown in Figure 3.4.

This node-level striping scheme does not require data replication and at the same time can balance the storage requirement equally among all nodes in the cluster. However, the system will need to coordinate all nodes in the cluster to participate in each and every video streams. We address this challenge in the next section.

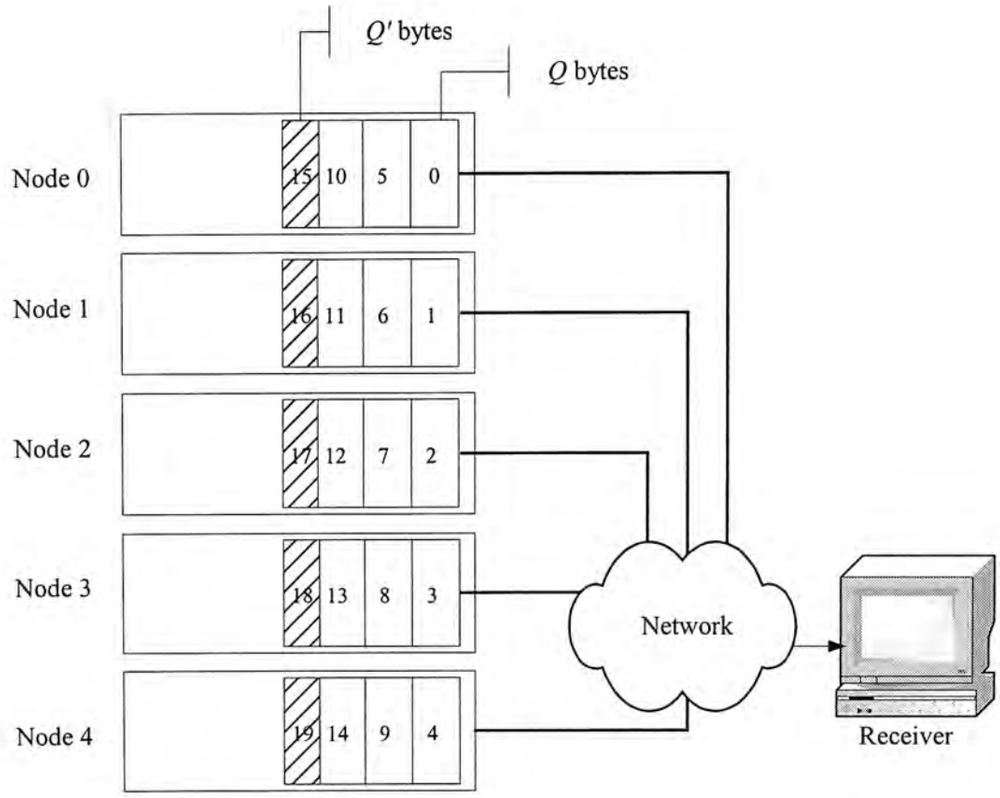

Figure 3.4. Storage imbalance solved by decreasing block size to Q' bytes in the final assignment round.

3.2 Retrieval and Transmission Scheduling

To initiate a video streaming session, a receiver node will first locate the set of sender nodes carrying blocks of the desired video title, the striping policy and other parameters (format, bitrate, etc.) through the directory service. The receiver node then sends requests directly to the sender nodes, using a reliable multicast transport protocol for example. The sender nodes upon receiving the request will begin retrieving video data for transmission to the receiver node. Assume all video titles are constant-bit-rate (CBR) encoded and share the same bitrate R_v bytes per second. Then a sender node will serve up to N video streams, of which N–1 of them are transmitted while the remaining one played back locally. As each video stream is served by all N

nodes concurrently, a sender node only needs to deliver a bitrate of R_v/N for each video stream.

As a sender node is similar to a mini video server, we can adopt existing video server schedulers for use in the sender nodes. Many existing video server designs employ round-based schedulers such as SCAN [19] and its variants. A more general version is the Grouped Sweeping Scheme (GSS) proposed by Yu, *et al.* [20]. Compared to the more common SCAN scheduler that maximizes throughput at the expense of buffer overhead, GSS allows one to control the tradeoff between disk efficiency and buffer requirement. This is a useful feature because a node will have relatively limited buffer capacity and GSS allows us to tradeoff between buffer requirement and disk throughput.

In GSS, streams are divided into g groups in which retrievals within a group are scheduled using SCAN. The groups are served in a round-robin manner as shown in Figure 3.5 for a GSS with three groups. We call the period of serving a group a *micro round* and the period of serving all groups once a *macro round*. If one set g=1 and g=N then GSS reduces to SCAN (higher throughput, larger buffer requirement) and FCFS (lower throughput, smaller buffer requirement) respectively. Intermediate values of g can be used to tradeoff between disk efficiency and buffer requirement.

Given a data block size of Q bytes, up to N/g data blocks will be retrieved in a micro round. These retrieved data blocks will then be transmitted at a rate of R_v/N for a duration equals to g micro rounds (i.e. a macro round). Let T_g and T_f be the micro round and macro round lengths respectively. We can then obtain them from

$$T_f = gT_g = \frac{NQ}{R_v} \tag{3.2}$$

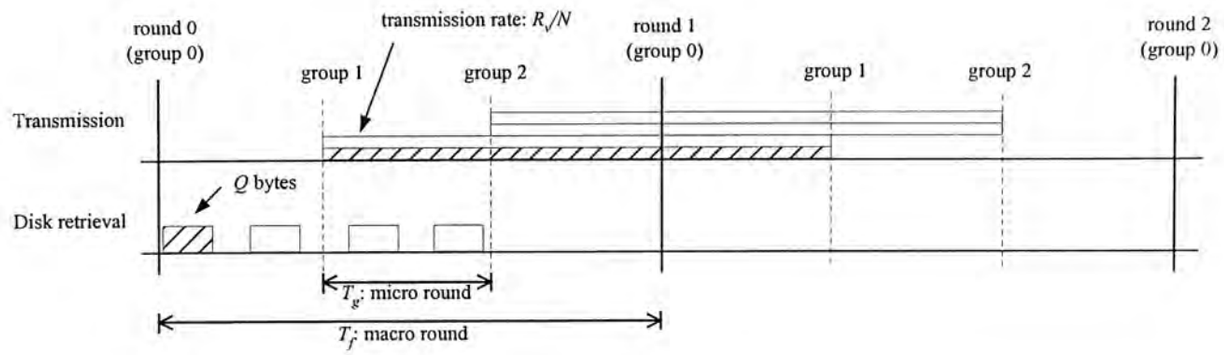

Figure 3.5. The Grouped Sweeping Scheme (shown with three groups, transmission is like fluid flow).

However, a problem arises when we employ the GSS scheduler in a distributed system. Specifically, as nodes run independently from each other in the system, their clocks are not precisely synchronized. Although there are distributed clock-synchronization protocols [21] that can be used to synchronize their internal clocks, slight deviation, called clock jitter, is inevitable. This may cause different nodes in the system to admit a video stream to different micro rounds in the GSS scheduler and eventually leads to inconsistent schedules and load balance problems. This problem has been previously investigated in the context of parallel video server [4]. Let δ be the maximum transmission time for a request. Assuming the clock jitter between any two nodes in the cluster is bounded by a constant, say τ, then we can prevent inconsistent scheduling by delaying the admission of all new video streams by

$$\Omega = \left\lceil \frac{(\tau + (N-1)\delta)\, g}{T_f} \right\rceil + 1 \tag{3.3}$$

micro rounds. Please refer to Appendix A for the proof and interested readers are referred to [4] for details of the original problem and the solution in the context of a parallel video server.

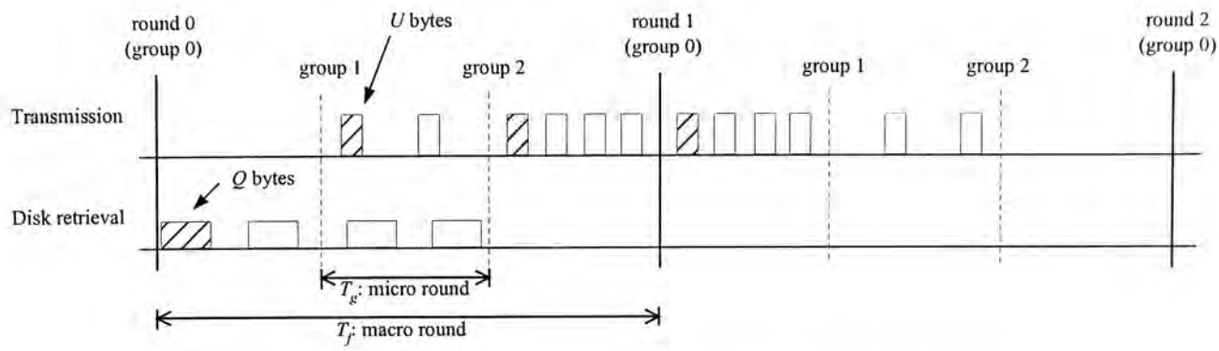

Figure 3.6. The GSS with transmission blocks for transmission.

Reconsidering the GSS model as depicted in Figure 3.5, there is an underlying assumption that masks another potential problem. In particular, the depicted model assumed that data are transmitted continuously like fluid flow. In practice, this is clearly not the case as data are usually transmitted in discrete units, i.e. packets, complete with header and other control information embedded. In the packetized transmission model, each retrieved block is divided into packets, or called transmission blocks, of size U bytes for transmission. That is,

$$Q = bU \tag{3.4}$$

where b is an integer. Figure 3.6 redraws the GSS scheduler incorporating this packetized transmission model with b=3. This packetized transmission model also reveals a challenge and an opportunity. We first consider the opportunity.

Specifically, as striping is done in units of Q bytes, a client receiving the U-bytes packets cannot playback the data until the block of b packets is completely received and this translates into buffer requirement. The packetized transmission model, however, suggests that we can reduce this buffer requirement simply by striping video data in units of U bytes instead of Q bytes as shown in Figure 3.7. In this case, the client can playback video data immediately after receiving packets 0 to 4 instead of having to wait until packets 0 to 14 are all received, thereby reducing the buffer requirement and startup latency accordingly.

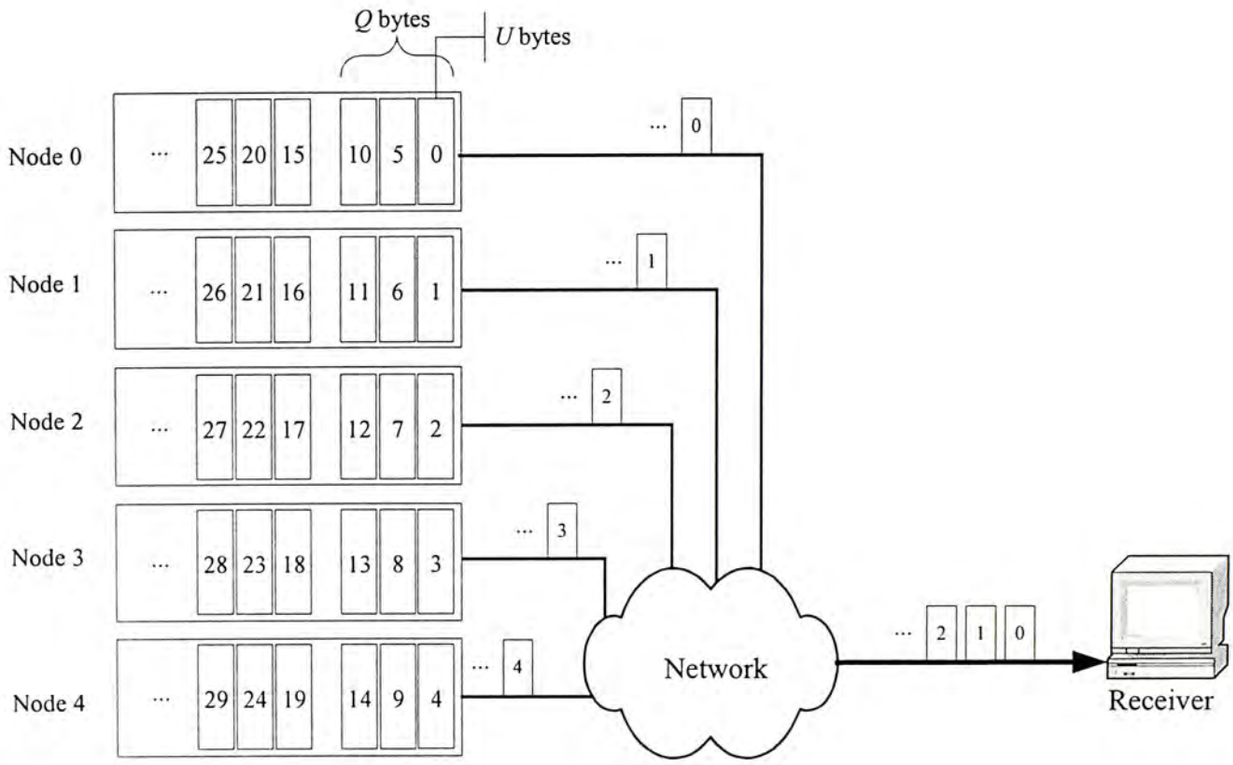

Figure 3.7. The video data is striped across the transmission block of size U bytes.

On the other hand, this packetized transmission model also creates a new challenge. In particular, if the nodes in the cluster transmit packets to the same receiver simultaneously, the resultant traffic burst will likely cause congestion when it reaches the client node as shown in Figure 3.8. To avoid this problem, we can stagger the transmission of packets destined to the same client node within a micro-round as depicted in Figure 3.9. The resultant traffic will then be smoothed out.

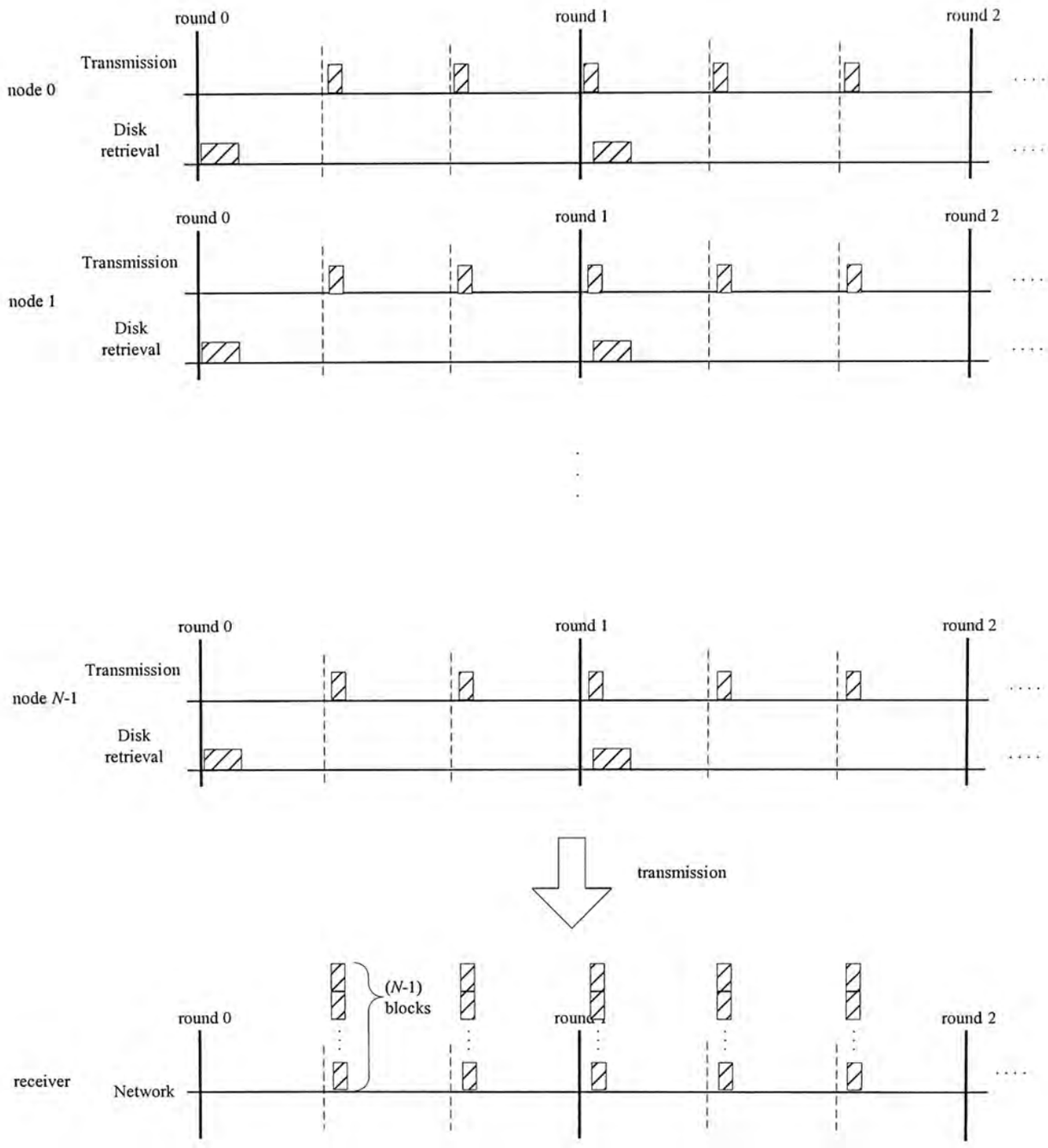

Figure 3.8. The receiver encounter traffic burst if all nodes transmit data packets at the same time

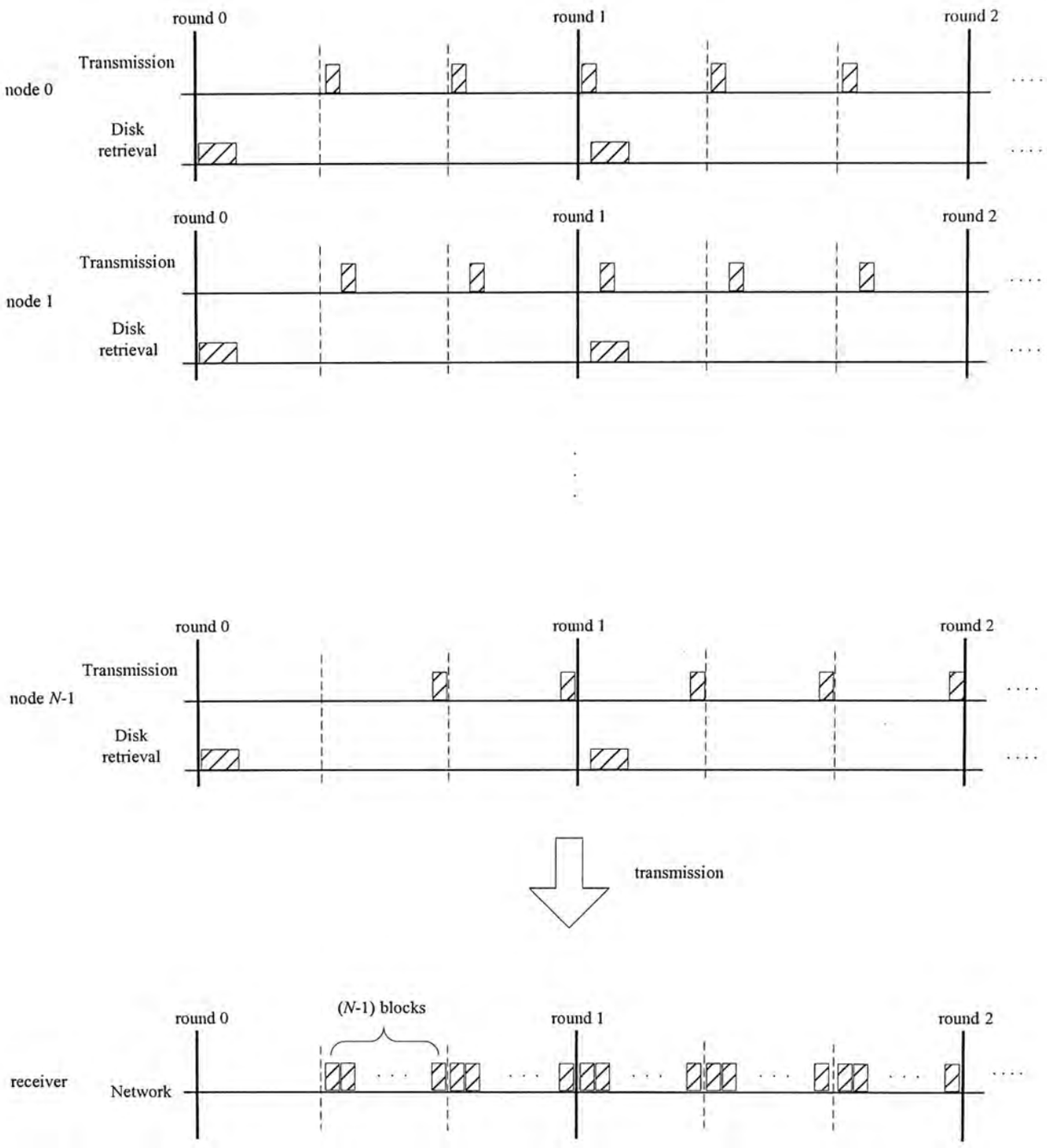

Figure 3.9. The receiver receives the transmission blocks in a staggered manner such that traffic bursts are avoided.

3.3 Fault Tolerance

In a server-less VoD system, fault tolerance becomes an essential capability as reliability of STBs and PCs is significantly lower than dedicated video servers located in a data center run by professional operators around the clock. Moreover, given the relatively large number of nodes, the system needs to expect and prepare to recover from not a single, but multiple simultaneously node failures.

When a node fails, all data stored in that particular node becomes unavailable. In communications terminology this is called data erasure. To recover from data erasures, erasure-correcting codes such as the Reed-Solomon Erasure Correcting (RSE) Code [22-23] can be used. Specifically, a (n, h)-RSE codeword comprises n symbols of which (n–h) of them are message symbols (i.e. data) and the remaining h are redundant symbols. One can recover all (n–h) message symbols as long as *any* (n–h) out of the n symbols are correctly received. Another paper of Rizzo [24] implemented the software erasure code encoder and decoder that can achieve the processing rate of several MB per second running on a Pentium 133 machine. With the current machines commonly equipped with Pentium III or 4 processors, they can provide sufficient processing power to perform erasure correction for playback.

By extending the striping-based placement policy in section 3.1 with a (N, h)-RSE code, the system will have sufficient redundant data for a receiver node to recover all video data with up to h simultaneous node failures in the cluster. To accommodate the RSE-code, we need to modify the placement policy and the schedulers. For the placement policy, an additional encoding step will be needed to compute the h redundant blocks for each group of (N–h) video data blocks. Moreover,

as now only (N–h) of the stored data are playable data, we will need to increase the striping block size from Q bytes to

$$Q_r = Q\left(\frac{N}{N-h}\right) \tag{3.5}$$

bytes to maintain the same amount of playable video data in a striping group. Hence the transmission block size also increases from U to U_r bytes.

$$U_r = \frac{Q_r}{b} = U\left(\frac{N}{N-h}\right) \tag{3.6}$$

For the disk scheduler, the retrieval block will be increased from Q bytes to Q_r bytes. Transmission rate will increase from R_v to

$$R_r = R_v\left(\frac{N}{N-h}\right) \tag{3.7}$$

to maintain the same video bitrate. Therefore incorporating redundancy incurs overhead in both disk retrieval and network transmission. We investigate in chapter 5 the amount of redundancy required to achieve a given system reliability.

Chapter 4

PERFORMANCE MODELING

In this chapter, we derive a performance model for the server-less architecture. In particular, we derive three key system requirements – storage requirement, bandwidth requirement, and buffer requirement; and one performance metric – system response time. We choose response time as the performance metric because it is the most visible performance metric while compared to storage, bandwidth and buffer requirement from the end-user perspective. For simplicity, we ignore network transmission delay and loss for new stream requests. In the case that the network transmission delay and loss cannot be ignored, we can increase the receiver buffer to tolerate the delay once the maximum and minimum delay are known, and increase the system redundancy level h to compensate for the transmission loss. Moreover, as discussed in chapter 3.3, today's machines can provide sufficient processing power to perform erasure correction for playback, so we also ignore the processing delay for simplicity.

4.1 Storage Requirement

Storage requirement measures how much disk storage per node is required to accommodate a given number of video titles. Under the striping-based placement policy presented in section 3.1, video data are striped across all nodes in the cluster

and so the storage requirement is divided equally among all nodes. It is also easy to see that the storage requirement is inversely proportional to the cluster size. Therefore in terms of storage requirement, larger cluster size is desirable.

Let S_A be the combined size of all video titles to be stored in the cluster, then the storage requirement per node, denoted by S_N will be given by

$$S_N = \frac{S_A}{N} \tag{4.1}$$

If a (N, h)-RSE code is employed in the placement policy, then additional storage will be needed to store the redundant data and the storage requirement increases to

$$S_R = \frac{N}{N-h} S_N = \frac{S_A}{N-h} \tag{4.2}$$

Given a set of video titles to be served in a cluster and the storage capacity of the nodes, the storage requirement then determines the lower limit on the cluster size.

4.2 Network Bandwidth Requirement

Network bandwidth usage reaches maximum when all nodes in the cluster stream videos simultaneously. Without redundancy, each node transmits video data to (N–1) other nodes in the cluster, with each stream consuming a bitrate of R_v/N bps. Therefore the aggregate network bandwidth requirement for a node without redundancy is

$$C = \frac{(N-1)}{N} R_v \tag{4.3}$$

If a (N, h)-RSE code is employed in the placement policy, then additional network bandwidth will be needed to send the redundant data and the per-node aggregate network bandwidth requirement increases to

$$C_R = C \frac{N}{N-h} = \frac{N-1}{N-h} R_v \tag{4.4}$$

With the node bandwidth requirement obtained, the network switching capacity then determines the upper limit on the cluster size because the required network switching capacity increases linearly with the cluster size.

4.3 Buffer Requirement

Two types of buffers are required in a node. First, sender buffers are required to store data being retrieved from the disk and for data being transmitted through the network. Second, receiver buffers are required for receiving data for video playback.

We first consider the sender buffer requirement. From the GSS study [20], we know that the sender buffer requirement is

$$B_s = \left(1+\frac{1}{g}\right)NQ \tag{4.5}$$

For the case of using a (N, h)-RSE code, the retrieval block size is increased from Q bytes to Q_r bytes. So replacing the Q with Q_r in the above expression, the sender buffer requirement for (N, h)-RSE code is given by,

$$\begin{aligned} B_{s,r} &= \left(1+\frac{1}{g}\right)NQ_r \\ &= \left(1+\frac{1}{g}\right)\frac{N^2}{(N-h)}Q \end{aligned} \tag{4.6}$$

Next, we derive the receiver buffer requirement. Using the approach similar to that in [4], we can obtain the receiver buffer requirement for the case of no redundancy. Interested readers may refer to Appendix B for the derivations.

$$B_r = 2\left(1+\left\lceil\frac{b\tau}{T_f}\right\rceil\right)NU \tag{4.7}$$

For the case with a (N, h)-RSE code employed for redundancy, an additional erasure correction step will need to be performed before data are ready for playback. With reference to section 3.3, one can recover all video data as long as at least (N–h)

out of the N blocks in a striping group are correctly received. This erasure correction process has two properties that affect buffer management. First, a client node cannot perform erasure correction until at least (N–h) blocks in a striping group are available. Second, having more than (N–h) blocks in a striping group offers no advantage to the erasure correction process.

Because of these two properties, we can use the same buffer size as in the case of no redundancy. To see why, first consider the buffer for receiving data from the network. The two properties imply that the client node only needs to receive (N–h) blocks for each striping group. Once (N–h) blocks are available, additional arriving blocks from the same striping group can be discarded because the erasure correction process will recover the original (N–h) blocks of video data regardless of which (N–h) blocks out of the striping group are received.

Now recall from (3.6) that the transmission block size under redundancy is increased to U_r. Then the buffer requirement will become

$$B_{r,r} = 2\left(1+\left\lceil\frac{b\tau}{T_f}\right\rceil\right)(N-h)U_r \tag{4.8}$$

Substituting U_r with U in (3.6) gives

$$\begin{aligned} B_{r,r} &= 2\left(1+\left\lceil\frac{b\tau}{T_f}\right\rceil\right)(N-h)\frac{N}{N-h}U \\ &= 2\left(1+\left\lceil\frac{b\tau}{T_f}\right\rceil\right)NU \end{aligned} \tag{4.9}$$

which is the same as the case without redundancy.

To obtain the total buffer requirement, we sum up the sender and receiver buffer requirement. However, as only (N-1) data streams are transmitted through network and the remaining one data stream is used for local playback, the buffer for

this local stream is double counted if it is considered separately in both sender and receiver buffer. Hence, b buffer blocks of size U bytes are subtracted from the receiver buffer to optimize the buffer requirement for the case without redundancy.

$$B_r = \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right)N - b \right] U \tag{4.10}$$

For the case with redundancy, subtracting b buffer blocks of size U_r from (4.8) gives,

$$\begin{aligned} B_{r,r} &= \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right)(N-h) - b \right] U_r \\ &= \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right)(N-h) - b \right] \frac{N}{N-h} U \\ &= \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right) - \frac{b}{N-h} \right] NU \end{aligned} \tag{4.11}$$

Combining (4.5) and (4.10), we can then obtain the total buffer requirement

$$\begin{aligned} B_t &= \left(1+\frac{1}{g}\right)NQ + \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right)N - b \right] U \\ &= \left[\left[\left(1+\frac{1}{g}\right) + \frac{2}{b}\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right) \right] N - 1 \right] Q \end{aligned} \tag{4.12}$$

for the case with no redundancy; and combining (4.6) and (4.11), we can obtain the total buffer requirement

$$\begin{aligned} B_{t,r} &= \left(1+\frac{1}{g}\right)\frac{N^2}{(N-h)}Q + \left[2\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right) - \frac{b}{N-h} \right] NU \\ &= \left[\frac{N}{(N-h)}\left(1+\frac{1}{g}\right) + \frac{2}{b}\left(1+\left\lceil \frac{b\tau}{T_f} \right\rceil\right) - \frac{1}{(N-h)} \right] NQ \end{aligned} \tag{4.13}$$

for the case with a (N, h) RSE code as redundancy.

4.4 System Response Time

System response time, denoted by T_s, comprises two components: scheduling delay, denoted by D_s, and prefetch delay, denoted by D_p:

$$T_s = D_s + D_p \tag{4.14}$$

Scheduling delay is defined as the average time from a client node requesting a new video session to the time data retrieval starts. Prefetch delay is defined as the average time from the start of data retrieval to the time when the prefetch buffer in the client node is filled with data ready for playback.

Under GSS with g groups, an arriving request may be admitted to the next upcoming micro round provided that it is not yet fully occupied, i.e., fewer than N/g streams are being served in that micro round. Otherwise it will need to scan through the upcoming list of micro rounds until it finds one that is available. Clearly the scheduling delay depends on the occupancy of the micro rounds, which in turn depends on the system utilization. This problem has been studied previously by Lee [4] in the context of parallel video server, who derived the average scheduling delay using an urn model. The equations are relatively lengthy and so will not be repeated here. Interested readers can find the complete derivation in Lee's work [4]. Summing up this average waiting time and the artificial delay (Ω) mentioned in section 3.2, we can then obtain the average scheduling delay under a given system utilization.

Now we consider the prefetch delay. Intuitively, the client needs to receive a stripe of transmission packets before performing erasure-correction to reconstruct the video data for playback. In practice, additional delay will be incurred due to clock jitter among nodes in the cluster. Incorporating this clock jitter into the model, it can be shown (see Appendix B) that the prefetch delay is given by

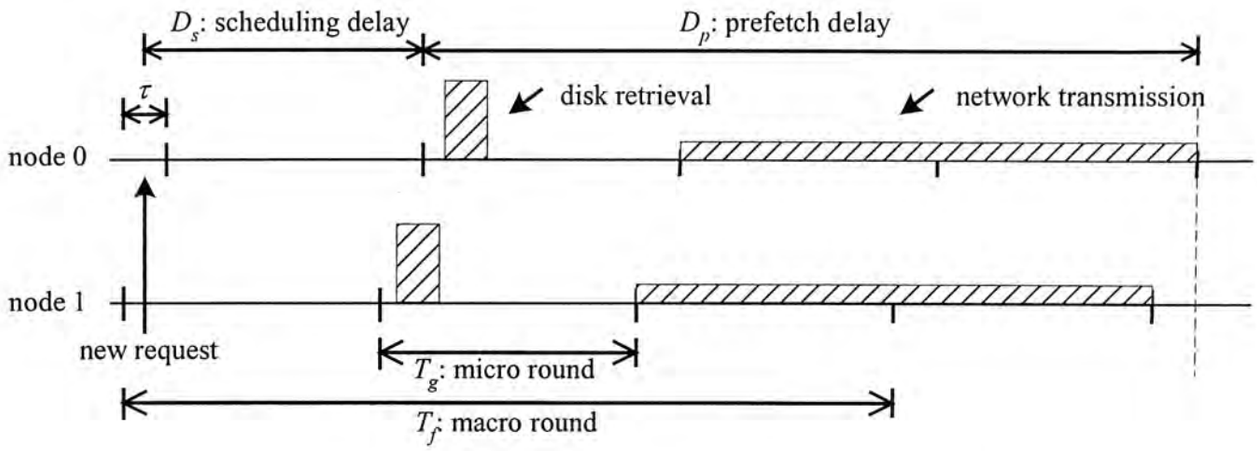

Figure 4.1. Composition of the system response time (g=3, b=3).

$$\begin{aligned} D_p &= T_g + \left(1 + \left\lceil \frac{b\tau}{T_f} \right\rceil\right)\frac{T_f}{b} \\ &= \left(\frac{1}{g} + \frac{1}{b}\left(1 + \left\lceil \frac{b\tau}{T_f} \right\rceil\right)\right)T_f \end{aligned} \tag{4.15}$$

Summing D_s and D_p will then give the system response time as shown in Figure 4.1.

Chapter 5

SYSTEM RELIABILITY

In section 3.3, we presented the use of a (N, h)-RSE code to implement node-level fault tolerance for the server-less VoD architecture. In this chapter we formulate a Markov chain model for the reliability of the system (Section 5.1). Next we investigate the amount of repair capability the system needs to achieve the target reliability with minimum redundancy (Section 5.2). Finally we determine the required redundancy level (i.e. parameter h) to meet the target reliability (Section 5.3) using conservative assumptions for the system parameters.

5.1 System Failure Model

We use mean time to failure (MTTF), defined as the average time between two consecutive occurrences of system failure, as the measure of the system's reliability. With the use of a (N, h)-RSE code, a system failure occurs when more than h out of the N nodes in the cluster fail. We assume nodes in the cluster fail independently with a node MTTF exponentially distributed with mean λ. A node, once failed, will be repaired immediately and independently. The repair time is also exponentially distributed with mean μ. We further define the redundancy level as the proportion of

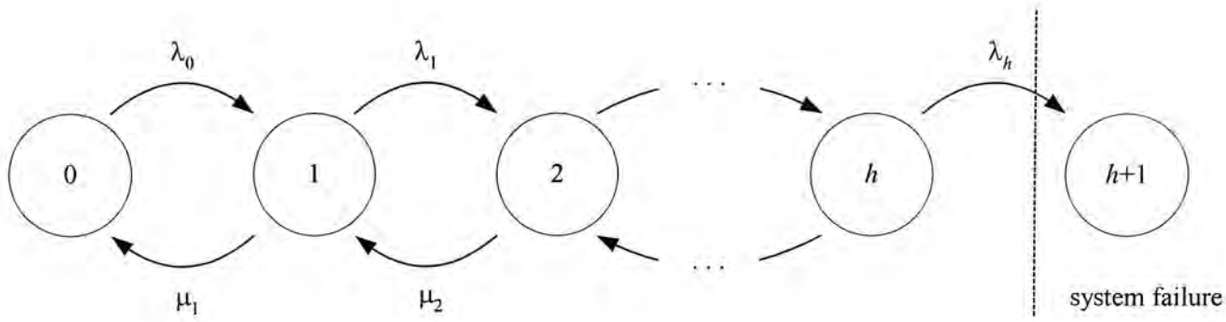

Figure 5.1. The Markov chain model for evaluating the system's mean time to failure.

nodes serving redundant data, i.e. (h/N). Our goal is to derive the relationship between the redundancy level and the system MTTF.

We model the system using a continuous-time Markov chain model as shown in Figure 5.1. Let state i be the system state where i out of the N nodes have failed. For example, state 0 means all nodes are operational, state 1 means one of them has failed, and so on. System failure occurs when the system enters state h+1, where the (N, h)-RSE code can no longer recover all (N–h) data blocks. We assume that once a system failure occurs, the system will be shutdown with all failed nodes repaired before the system is restarted.

In this Markov chain model, the system MTTF is equivalent to the expected time for the system to transit from state 0 to state h+1, or called the first passage time. Let λ_i be the rate at which the system transits from state i to state i+1 and μ_i be the rate at which the system transits from state i to i–1. We can obtain λ_i and μ_i from

$$\lambda_i = (N - i)\lambda \tag{5.1}$$

$$\mu_i = i\mu \tag{5.2}$$

Let T_i be the expected time the system takes to reach state h+1 from state i. Starting from state 0, the expected time to reach state h+1 is equal to the expected time to reach state 1 from state 0, i.e., equal to $1/\lambda_0$, plus the expected time to reach state h+1 from state 1, i.e., T_1 by definition. Thus we can obtain the following relation:

$$T_0 = \frac{1}{\lambda_0} + T_1 \tag{5.3}$$

Next we consider state 1. The system may either transit to state 2 with probability ($\lambda_1/(\lambda_1+\mu_1)$), or transit to state 0 with probability ($\mu_1/(\lambda_1+\mu_1)$). The combined transition rate is equal to ($\lambda_1+\mu_1$) and so the expected time until transition occurs is equal to $1/(\lambda_1+\mu_1)$. Using argument similar to the case of state 0, we can obtain another relation:

$$T_1 = \frac{1}{\lambda_1+\mu_1} + \frac{\lambda_1}{\lambda_1+\mu_1} T_2 + \frac{\mu_1}{\lambda_1+\mu_1} T_0 \tag{5.4}$$

Repeating this procedure for T_2 to T_{h+1}. We can obtain the following set of equations:

$$\begin{aligned}
&T_1 = \frac{1}{\lambda_1+\mu_1} + \frac{\lambda_1}{\lambda_1+\mu_1} T_2 + \frac{\mu_1}{\lambda_1+\mu_1} T_0 \\
&\vdots \\
&T_h = \frac{1}{\lambda_h+\mu_h} + \frac{\lambda_h}{\lambda_h+\mu_h} T_{h+1} + \frac{\mu_h}{\lambda_h+\mu_h} T_{h-1} \\
&T_{h+1} = 0
\end{aligned} \tag{5.5}$$

Substitute (5.3) into (5.4), we can eliminate the term T_0 and express T_1 in terms of T_2 only. Repeating this substitution recursively, we can simplify the above equations into the following equations:

$$\begin{aligned}
&T_1 = \frac{1}{\lambda_1} + \frac{\mu_1}{\lambda_1\lambda_0} + T_2 \\
&T_2 = \frac{1}{\lambda_2} + \frac{\mu_2}{\lambda_2\lambda_1} + \frac{\mu_2\mu_1}{\lambda_2\lambda_1\lambda_0} + T_3 \\
&T_3 = \frac{1}{\lambda_3} + \frac{\mu_3}{\lambda_3\lambda_2} + \frac{\mu_3\mu_2}{\lambda_3\lambda_2\lambda_1} + \frac{\mu_3\mu_2\mu_1}{\lambda_3\lambda_2\lambda_1\lambda_0} + T_4 \\
&\vdots \\
&T_i = \left(\sum_{j=0}^{i} \frac{\prod_{k=0}^{j-1} \mu_{i-k}}{\prod_{k=0}^{j} \lambda_{i-k}} \right) + T_{i+1} \\
&\vdots \\
&T_{h+1} = 0
\end{aligned} \tag{5.6}$$

As T_{h+1} is zero, we can obtain the value of T_0 by backward substitution:

$$T_0 = \sum_{i=0}^{h} \left(\sum_{j=0}^{i} \frac{\prod_{k=0}^{j-1} \mu_{i-k}}{\prod_{k=0}^{j} \lambda_{i-k}} \right) \tag{5.7}$$

which is the MTTF of the system.

5.2 Minimum System Repair Capability

The failure model in the previous section assumes that the repair capability of the system is unlimited. In other words, all failed nodes are repaired immediately and simultaneously. In practice, the repair rate is usually limited. For example, if repairs are done manually by repairmen, then the number of repairmen is likely to be limited and also much smaller than the number of nodes in the cluster. To account for this constraint, we extend the failure model in the previous section to model bounded repair rate.

Let m be the system repair rate limit such that all repair rates in the Markov chain model are bounded by $m\mu$. Using the repairmen example, this represents a total of m repairmen available for repairing failed nodes in the cluster. The repair rate for state i is then changed to

$$\mu_i = \begin{cases} i\mu & i < m \\ m\mu & i \geq m \end{cases} \tag{5.8}$$

We call the ratio m/N the system's repair capability, which can range from 0 (i.e., no repair available) to 1 (i.e. no constraint on maximum repair rate). Substituting the repair rate in (5.8) into (5.6) and (5.7), we can then obtain the corresponding system MTTF with limited system repair capability. Clearly, in this case we will need to increase the redundancy level to achieve the same target system MTTF.

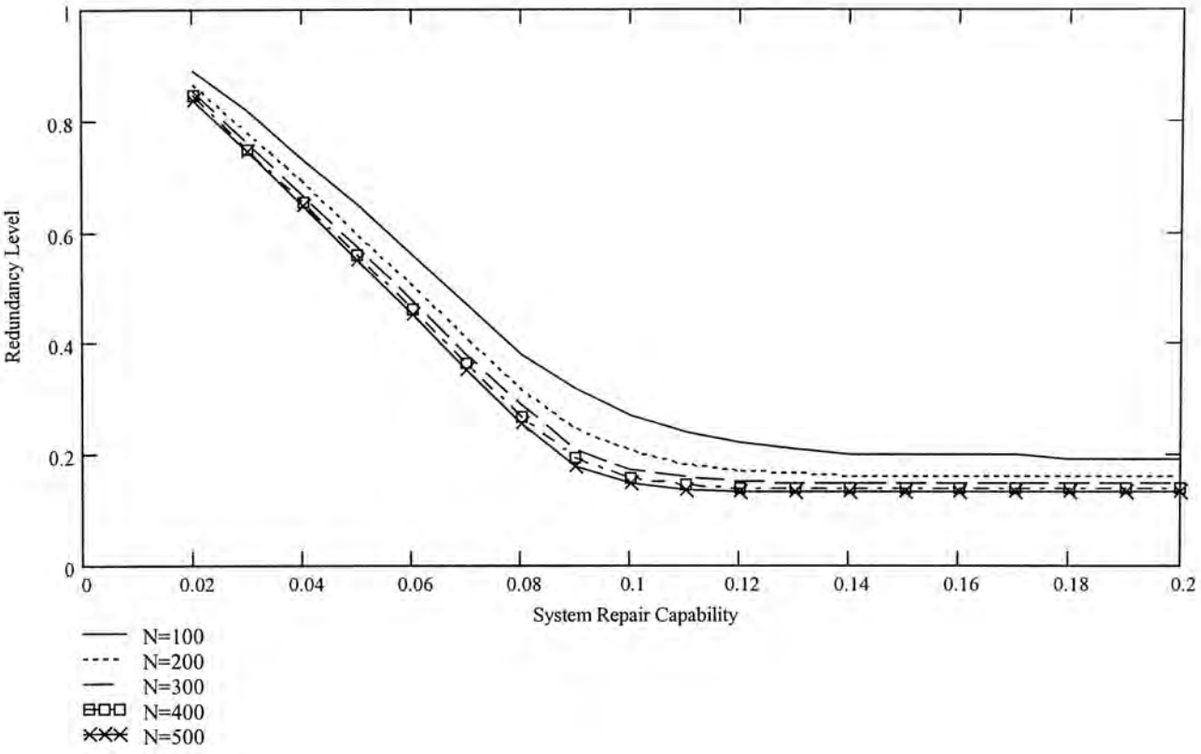

Figure 5.2. The redundancy level required to achieve a system MTTF of 10000 hours under different repair rate limit.

Figure 5.2 plots the redundancy level required to achieve system MTTF of 10,000 hours versus system repair capability ranging from 0.02 to 0.2. We observe that there is a turning point in the curves, approximately around a system repair capability of 0.1. Below which the redundancy overhead increases with lower system repair capability; and above which the redundancy overhead stays relatively constant even when the system repair capability is increased. The reason leading to this turning point is the ratio between node MTTR and node MTTF.

In particular, we have used a node MTTR of 24 hours and a node MTTF of 256 hours in computing the results in Figure 5.2. This MTTR/MTTF ratio is equal to 0.09375, close to the system repair capability at the turning point in Figure 5.2. These system parameters imply that a node repairs itself around ten times faster than the tendency to fail. Hence with a system repair capability of 0.1, or $10m=N$, the aggregate repair rate of $0.1N$ nodes will be comparable to the aggregate failure rate of

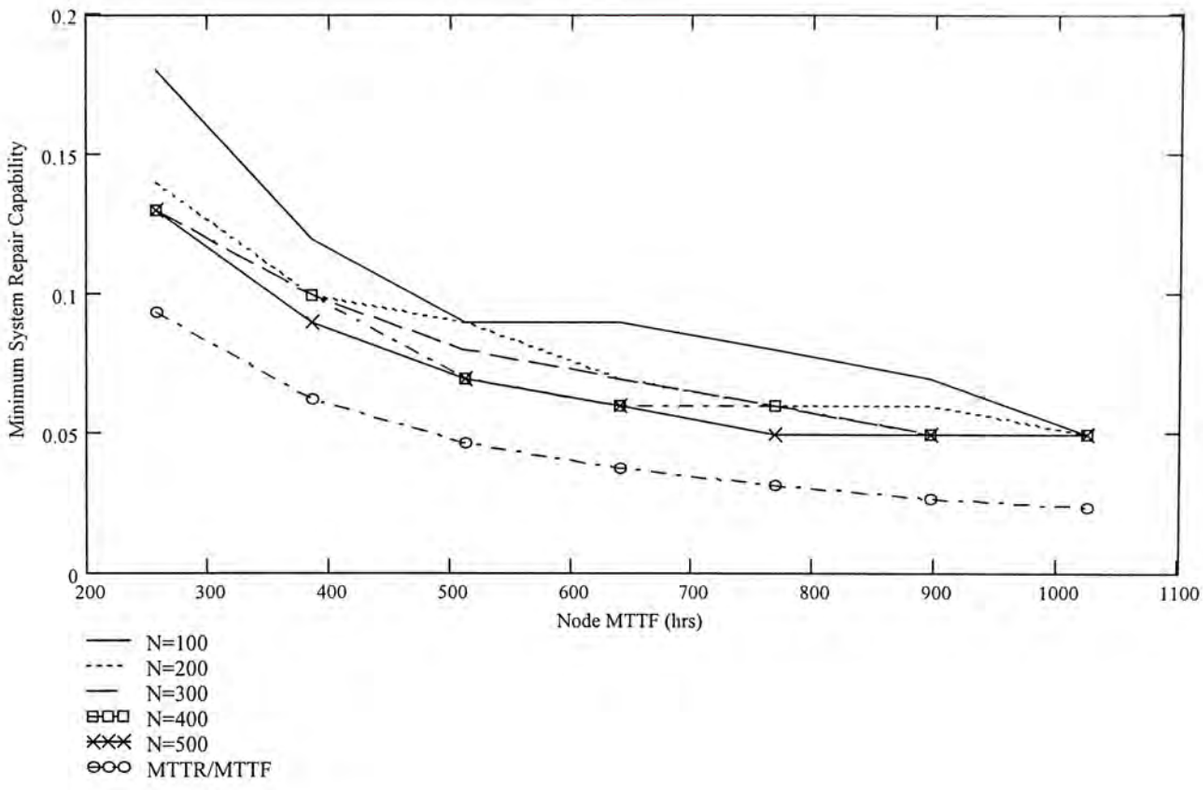

Figure 5.3. The minimum system repair capability required versus the node MTTF at fixed cluster sizes.

all N nodes. In other words, beyond this turning point the system will be able to repair nodes faster than they fail.

A consequence of this result is that it will not be necessary to increase the system repair capability far beyond the turning point as the reduction in redundancy overhead will be negligible. Figure 5.3 plots such turning points versus node MTTF ranging from 256 hours to 1,024 hours. The turning points, labeled minimum system repair capability in Figure 5.3, is obtained by increasing the system repair capability, i.e., increasing m, until the redundancy overhead reduces to equal to the case of unlimited system repair capability.

The results show that a system repair capability ranging from 38.67% to 161.33% over than MTTR/MTTF ratio will be sufficient. Moreover, the additional repair capability needed decreases with increases in cluster size. For example, the

additional repair capability needed is 92% for a 100-node cluster with node MTTF and MTTR equal to 256 hours and 24 hours respectively, but decreases to 38.67% for a 500-node cluster. Increasing the cluster size further will make the needed additional repair capability asymptotically approaches the MTTR/MTTF ratio.

5.3 Redundancy Configuration

In this section, we investigate the minimum redundancy configuration needed to achieve server-grade MTTF. Client-size devices such as STBs and PCs are inherently less reliable. Therefore while servers typically have MTTF over 10,000 hours, we assume conservatively a node MTTF of 256 hours only. Once a node fails, repair will be performed immediately. In practice, the repair time will depend on the type of failure occurred. For example, a power surge may cause a node to reboot itself and the repair time will be in minutes. However, if the failure is due to component failure such as a harddrive failure or memory failure, then the repair time will be much longer. As no data exists for the repair rate of STBs, we assume that the node MTTR is 24 hours. Using results from the previous section, we configure the system with the minimum system repair capability such that redundancy overhead is minimized. In case the system has lower repair capability, it can still achieve the target system MTTF with the use of additional redundancy (cf. Section 5.2).

Figure 5.4 plots the redundancy level required to achieve a system MTTF of 1,000, 10,000 and 100,000 hours versus cluster size. There are two observations. First, the redundancy level required to achieve a given system MTTF decreases with larger cluster size, implying that larger cluster sizes are desirable from a reliability point of view. Second, we observe that the redundancy level required to achieve a MTTF of 10,000 hours (i.e. over one year) or even 100,000 hours (i.e. over eleven years) are

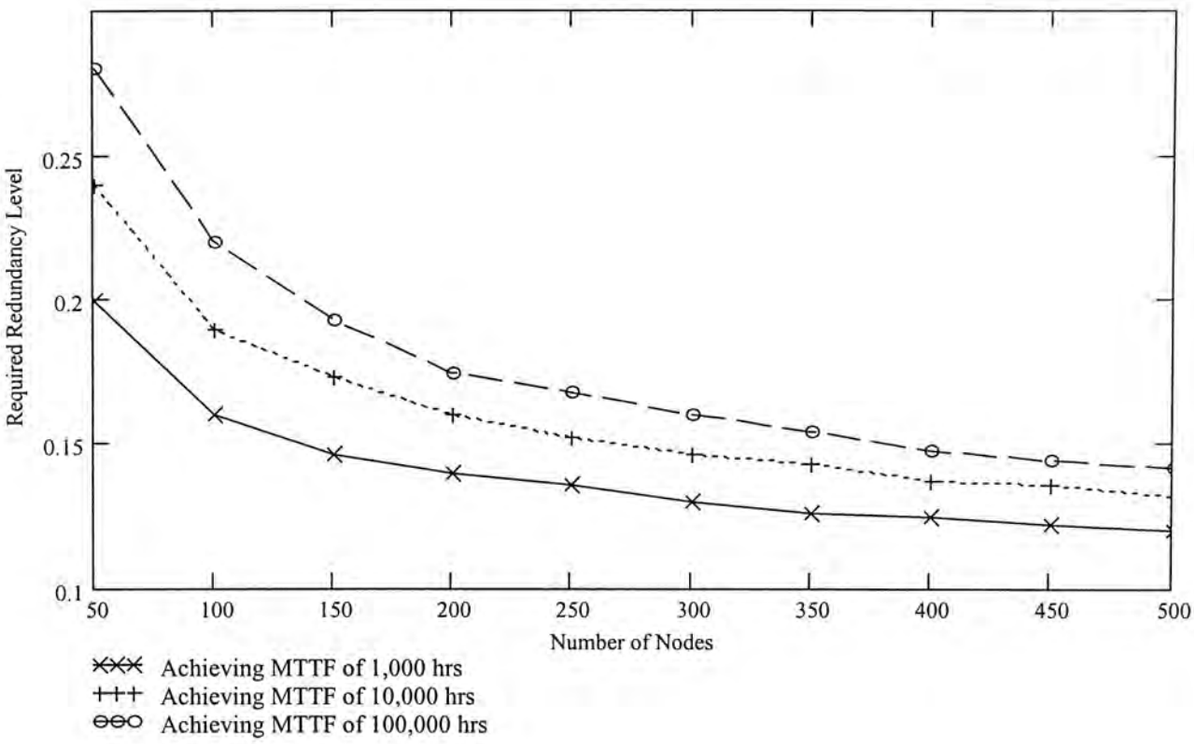

Figure 5.4. Required redundancy level versus cluster size.

relatively modest, at 0.132 and 0.142 respectively for a 500-node cluster. Despite the modest redundancy, this level of system MTTF already exceeds those of dedicated video servers. In the next chapter, we study other resource requirements and the scalability of the server-less architecture.

Chapter 6

SYSTEM DIMENSIONING

In system dimensioning, we study how large we can scale up a cluster in the system and what is the limiting factor to scalability. Under the server-less architecture, we obviously no longer have the dedicated server to become the bottleneck of the system. Hence, the limiting factor either arises from the nodes, or the network that links up the nodes. We consider the storage limit in section 6.1, network limit in section 6.2, and then investigate node-related bottlenecks in section 6.3-6.5. The system parameters are listed in Table 1.

Table 1. System parameters used in chapter 6.

Parameters	Symbol	Value
Node mean time to failure	$1/\lambda$	256 hours
Node mean time to repair	$1/\mu$	24 hours
Video bitrate	R_v	4 Mb/s
Disk fixed overhead	α	0.176 ms
Disk rotational latency	W^1	5.99 ms
Worst-case disk transfer rate	r_{min}	18.68 MB/s
Transmission time of notification packet	δ	54.4μs
Clock jitter	τ	100 ms

6.1 Storage Capacity

Under the striping-based placement policy, all nodes in the cluster equally share the storage requirement. Hence, the more nodes in a cluster, the less storage is required at each node to store the same amount of video data. Therefore the storage requirement imposes a lower limit on the scale of a cluster.

For example, assuming a video bitrate of 4 Mbps (e.g. MPEG-2 video) and a video length of 2 hours (e.g. movies), the total storage for 100 videos is 351.6 GB. Given that today's harddisks have at least several tens of gigabytes of storage capacity, even if each node only allocates 2 GB for video storage, the minimum cluster size needed is still only 209 nodes with redundancy level of 0.158 to achieve a system MTTF of 10,000 hours.

6.2 Network Capacity

One way to connect nodes together to form a cluster is to use a switch-based network such as switched Ethernet. Today's medium-range Ethernet switches typically has switching capacity of 32 Gbps or more while carrier-class switches easily exceeds 192 Gbps. Assuming a video bitrate of 4 Mbps and ignoring protocol overhead, a node will consume 3.98 Mbps both upstream and downstream without redundancy, and 4.73 Mbps with a redundancy level of 0.158 for a cluster size of 209 nodes to achieve a system MTTF of 10,000 hours. For larger cluster sizes, the per-node network bandwidth requirement will decrease as less redundancy is required to achieve the same system MTTF (cf. Section 5.3). Suppose we use 4.73 Mbps as the upper bound on bandwidth consumption for cluster size larger than 209 nodes. Then with a 32 Gbps switch running at 60% utilization, this translates into a maximum

cluster size of 2,029 nodes. Upgrading it to a 192 Gbps switch will extend the limit to 12,174 nodes.

On the other hand, the server-less architecture does require more bandwidth for the last-mile access network. In particular, the architecture requires more upstream bandwidth than traditional client-server-based VoD systems, which require little to no upstream bandwidth. This requirement will rule out some access network with asymmetric bandwidth such as ADSL or cable modem. Nevertheless, the emerging metropolitan networks built upon Ethernet-based networks running at 10Mbps or even 100Mbps full-duplex will provide more than sufficient bandwidth to accommodate a server-less VoD system.

6.3 Disk Access Bandwidth

A node obviously will have finite resources, including memory and disk access bandwidth. We first investigate the disk access bandwidth issue as it determines the configuration of the disk scheduler (GSS), which in turn determines the buffer requirement.

The disk scheduler has two configurable parameters, namely block size Q_r and the number of groups g in GSS. Increasing the block size or decreasing the number of groups in GSS results in higher disk efficiency, at the expense of larger buffer requirement. Therefore in terms of buffer requirement, it is desirable to reduce the block size Q_r and to increase the number of groups g in GSS.

Now consider the disk model. The time for serving a request retrieval, denoted by $t_{request}$, is composed of four components [20], namely fixed overhead (e.g. head-switching time) denoted by α, seek time denoted by t_{seek}, rotational latency denoted by $t_{latency}$ and a transfer time equal to Q_r/r, where r is the disk transfer rate:

$$t_{request}(Q_r) = \alpha + t_{seek} + t_{latency} + \frac{Q_r}{r} \tag{6.1}$$

Under GSS, retrieval requests are divided into g groups, each served in a micro round using the SCAN scheduler. Under the worst-case scenario, the maximum time to retrieve k requests in a micro round can be computed from

$$t_{round}(k, Q_r) = k\alpha + t_{seek}^{\max}(k) + k\left(W^{-1} + \frac{Q_r}{r_{\min}}\right) \tag{6.2}$$

where $t_{seek}^{\max}(k)$ is the worst-case combined seek time for k requests, W^1 is the time for one complete round of disk platter rotation, and r_{min} is the minimum disk transfer rate[1].

To ensure continuous video streaming, retrievals for these k requests must be completed within the duration of a micro round [25]. Therefore the following condition must be satisfied:

$$t_{round}\left(\frac{N}{g}, Q_r\right) \leq \frac{T_f}{g} \tag{6.3}$$

where t_{round} represents the worst-case disk service round length.

For simplicity, we assume that the disk is reserved for use by the VoD application and is not shared with other applications. The disk model can be extended to support other concurrent disk accesses, for example, by reserving disk bandwidth in each micro round for such applications. Using this disk model, we compute the buffer requirement and system response time in the following sections.

---

[1] Modern disks commonly use disk zoning, i.e., varying track size, to increase storage capacity. This results in varying disk transfer rate depending on the zone. r_{min} refers to the lowest transfer rate among all zones.

6.4 Buffer Requirement

To determine the buffer requirement and the system response time (Section 6.5), we have to first determine the retrieval block size (Q_r), the transmission block size (U_r), the redundancy level (h), and the number of GSS groups (g). As the sender buffer requirement is directly proportional to the retrieval block size (4.6), we should reduce the retrieval block size (Q_r) as long as the continuous streaming condition is still satisfied (6.3). We use an off-the-shelve harddisk – Quantum Atlas 10K [26] as an example for computing numerical results.

For the transmission block size (U_r), a smaller transmission block size can reduce the receiver buffer requirement at the expense of increased header overhead. To maintain header overhead to a small level, we set the transmission block size to be not smaller than 8KB. We then find the minimum system response time achievable and the corresponding buffer requirement by evaluating all combinations of Q_r, U_r and g.

Table 2 lists the minimum system response time, the corresponding buffer requirement, and the optimized system configurations. Redundancy is already included to maintain a system MTTF of 10,000 hours (cf. Section 5.3). A surprising finding is that the optimized retrieval block size is equal to the transmission block size, at the minimum size of 8KB. This is in sharp contrast to the conventional video server design principle that makes use of large retrieval block size to increase disk efficiency. In other words, disk bandwidth is more than sufficient in a server-less VoD system due to the distribution of streaming bandwidth requirement to all nodes in the cluster.

Table 2. System configurations optimized for minimum system response time.

Cluster Size	Redundancy	Group	Block Size (Q_r bytes)	Macro-round (sec)	Buffer (MB)	Sys Resp Time (sec)
25	8	5	9216	0.2988	0.8525	0.8965
50	12	10	8192	0.5938	1.6094	1.5141
75	16	25	8192	0.9219	2.4453	2.1242
100	19	20	8192	1.2656	3.3438	2.8929
125	23	25	8192	1.5938	4.1953	3.5508
150	26	50	8192	1.9375	5.0625	4.1846
175	29	35	8192	2.2813	5.9609	4.9386
200	32	40	8192	2.6250	6.8438	5.6332
225	35	75	8192	2.9688	7.7109	6.2542
250	38	50	8192	3.3125	8.6094	7.0135
275	41	55	8192	3.6563	9.4922	7.7008
300	44	100	8192	4.0000	10.3594	8.3244
325	47	65	8192	4.3437	11.2578	9.0791
350	50	70	8192	4.6875	12.1406	9.7698
375	52	125	8192	5.0469	13.0391	10.4205
400	55	200	8192	5.3906	13.9141	11.0946
425	58	85	8192	5.7344	14.8203	11.8658
450	61	225	8192	6.0781	15.6797	12.4707
475	63	95	8192	6.4375	16.6172	13.2745
500	66	250	8192	6.7813	17.4766	13.8786

Figure 6.1 plots the sender, receiver, and total buffer requirements for cluster size up to 500. The results show that the buffer requirements all increase with the cluster size but the receiver buffer requirement dominates the total buffer requirement. Nevertheless, given the current cost of memory, the total buffer requirement, while not insignificant, should be practical for STBs and ordinary PCs.

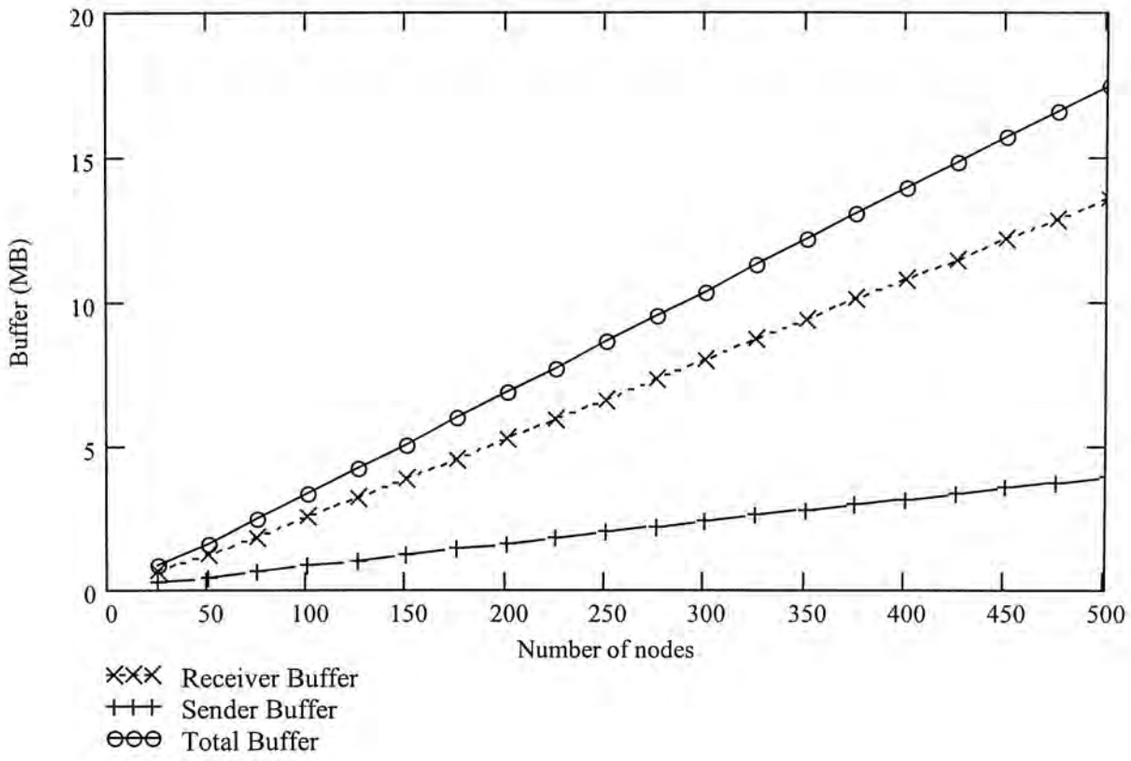

Figure 6.1. Buffer requirements versus cluster size.

6.5 System Response Time

To compute the response time, we assume a maximum request packet transmission time (cf. Section 3.2) of δ=54.4 μs. This is computed from the assumption of sending a 68-byte request packet using UDP over 10Mbps Ethernet, with 26 bytes of Ethernet frame header, 20 bytes of IP header, 8 bytes of UDP header and 14 bytes of protocol header (including 4 bytes of movie id, 2 bytes of receiver reception port, 4 bytes of scheduled group number and 4 bytes of control message).

Assuming 90% system utilization and using values of Q_r and g that minimize system response time, we plot in Figure 6.2 the system response time, the scheduling delay, and prefetch delay for cluster sizes up to 500. The results clearly show that the system response time is dominated by the prefetch delay, which increases near linearly with cluster sizes while the scheduling delay stays relatively constant.

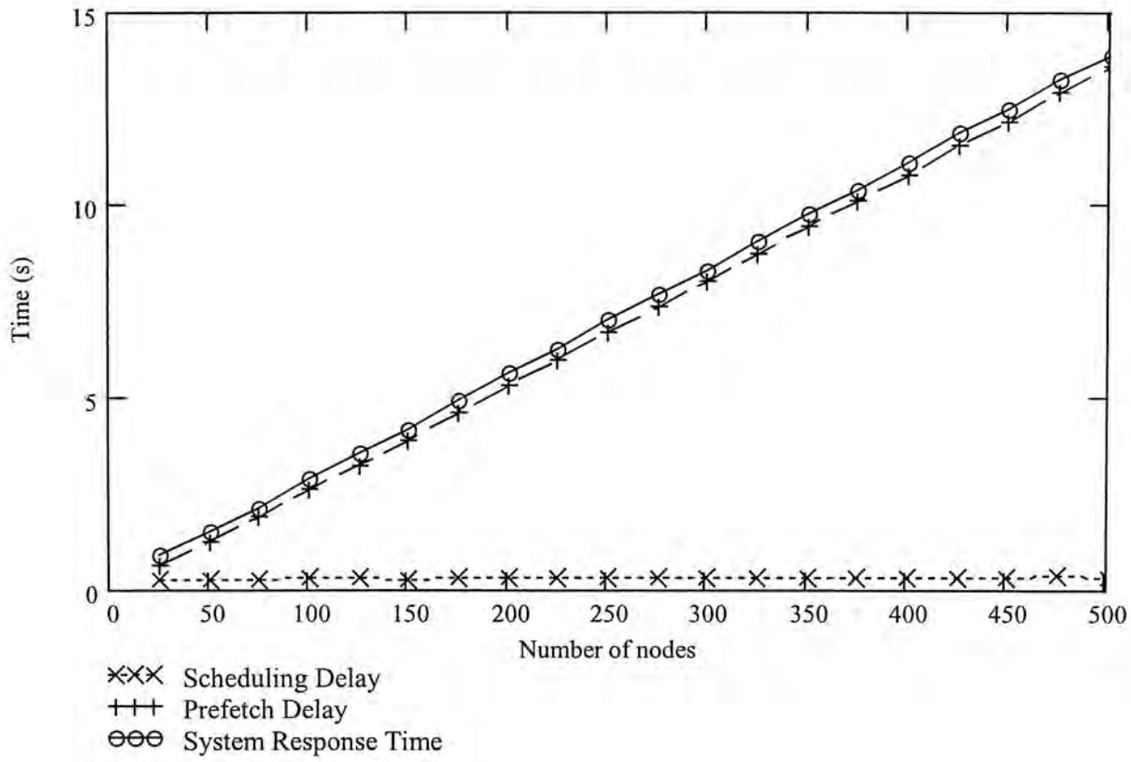

Figure 6.2. System response time, scheduling delay, and prefetch delay versus cluster size (90% utilization).

Another observation is that the system response time can become the performance bottleneck at larger cluster sizes. For example, a 500-node cluster has a system response time of 13.88 seconds, bordering on unacceptable for interactive VoD applications. We tackle this problem in the next chapter by a simple yet effective solution.

Chapter 7

MULTIPLE PARITY GROUPS

The results in the previous chapter reveals that the primary limit on the scalability of a cluster is, surprisingly, the system response time. Reconsidering Figure 6.2, we can observe that the prefetch delay is the key bottleneck. Prefetch delay in turn, depends on the amount of data a client node must receive before playback can begin. Due to the use of erasure-correction code, a client node must receive a minimum number of transmission packets from a parity group before erasure correction can be performed, and the corrected video data can be played back. In a N-node cluster with a redundancy level of h, this minimum number is equal to $(N-h)$, which clearly increases with the cluster size N.

This motivates us to investigate breaking down the cluster into multiple smaller parity groups that are independently erasure-coded. In this way, the minimum number of transmission packets a client must receive before playback will be reduced. Specifically, we divide the cluster into p parity groups with each group containing N/p nodes[2]. Each parity group i, (i=0,1,...,p–1), is then independently coded using an erasure-correction code with a redundancy level of h_i. For the case that N is divisible

---

[2] If N is indivisible by p, then (N mod p) out of p groups contain $\lceil N/p \rceil$ nodes while the remaining groups contain $\lfloor N/p \rfloor$ nodes.

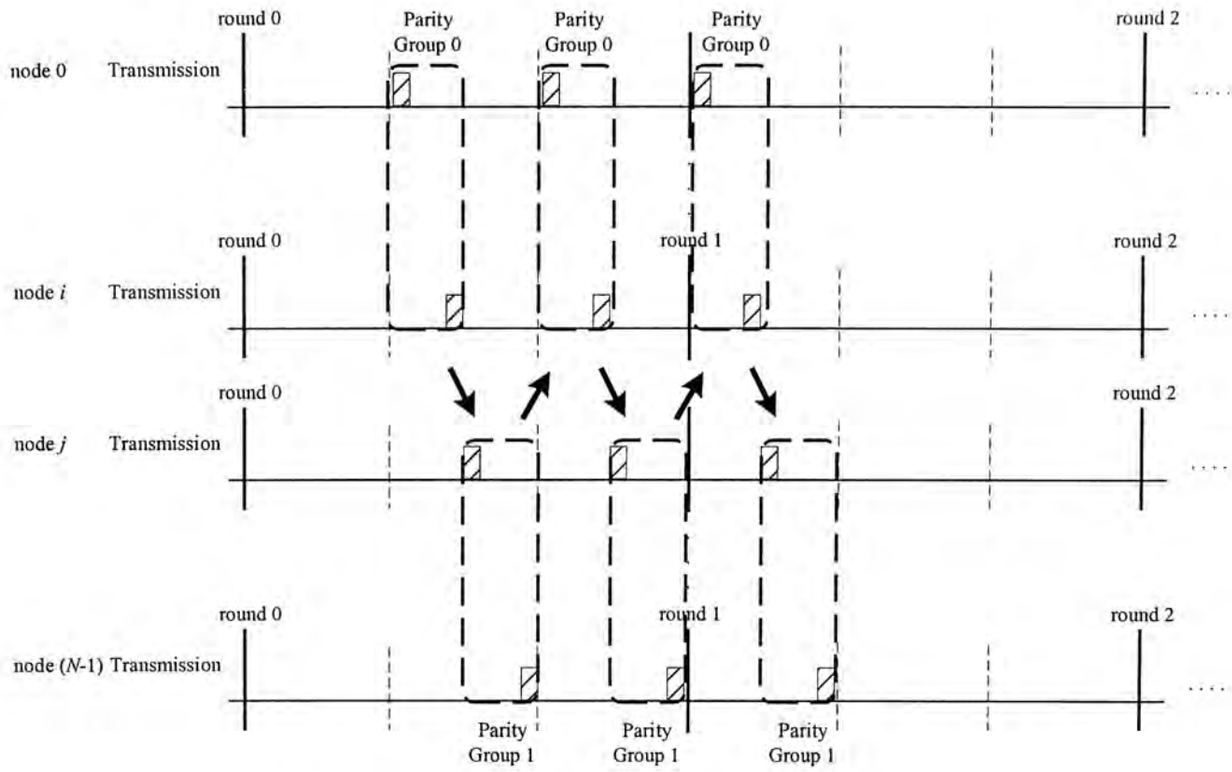

Figure 7.1. An example of two parity groups.

by p, the redundancy levels of all parity groups are same and we denote it by h_p.

One surprising property with this multiple parity group scheme is that it can be implemented without modification to the retrieval and transmission scheduler. As shown in Figure 7.1, the parity groups happen to be time-staggered in the transmission schedule due to the packet-staggering scheme originally introduced in section 3.2 for preventing network congestion. In this way, the client can immediately begin playback after receiving $(N/p-h_p)$ transmission packets from parity group 0 and thus significantly reducing the prefetch delay.

There is, however, one tradeoff with the multiple parity group scheme – redundancy overhead. Recalling Figure 5.4 in section 5.3, we need more redundancy to maintain the same system MTTF in smaller clusters (i.e. smaller parity groups). Therefore dividing a large parity group into multiple smaller parity groups will increase the redundancy overhead. We formulate in the next section a new system

failure model incorporating multiple parity groups and then evaluate the impact on buffer requirement, system response time, and ultimately scalability in the subsequent sections.

7.1 System Failure Model

In cluster with a single parity group, a system failure occurs when more than h out of the N nodes in the cluster fail simultaneously. In cluster with multiple parity groups, the system fails when *any one* of the p parity groups fails. We assume the system will be shutdown with all failed nodes repaired whenever a system failure occurs.

Let T_p be the target MTTF of all parity groups, which can be computed from (5.1)-(5.7) with the cluster size N and the redundancy h replaced by the corresponding parity group size and group redundancy h_i respectively. Then the system MTTF, denoted by T_{sys}, is given by

$$T_{sys} = \frac{T_p}{p} \tag{7.1}$$

as the system failure rate is p times the failure rate of a parity group. The system redundancy level is then the sum of the redundancy level of all parity groups:

$$h = \sum_{i=0}^{p-1} h_i \tag{7.2}$$

We investigate the performance impact of this new system failure mode in the following sections.

7.2 Buffer Requirement

Derivation of the sender buffer requirement is the same as in section 4.3 as the retrieval and transmission scheduler is not changed. Following the sender buffer

requirement equation in (4.6), we can obtain the new sender buffer requirement for a p-parity-group cluster:

$$B_{s,p} = \left(1+\frac{1}{g}\right)\frac{N^2}{(N-h)}Q \tag{7.3}$$

For the receiver buffer requirement for a p-parity-group cluster, we note that the minimum number of transmission packets that must be received before playback is reduced from (N–h) to (N/p–h_p), and the time to receive these (N/p–h_p) transmission packets is equal to T_f/bp. Thus, the new receiver buffer requirement is given by

$$\begin{aligned} B_{r,p} &= 2\left(1+\left\lceil\frac{\tau}{T_f/bp}\right\rceil\right)\left(\frac{N}{p}-h_p\right)U_r \\ &= 2\left(1+\left\lceil\frac{\tau}{T_f/bp}\right\rceil\right)\left(\frac{N-h}{p}\right)\frac{N}{N-h}U \\ &= 2\left(1+\left\lceil\frac{bp\tau}{T_f}\right\rceil\right)\frac{NU}{p} \end{aligned} \tag{7.4}$$

Summing up the sender and receiver buffer requirement, we can then obtain the total buffer requirement:

$$\begin{aligned} B_{t,p} &= \left(1+\frac{1}{g}\right)\frac{N^2}{(N-h)}Q + 2\left(1+\left\lceil\frac{bp\tau}{T_f}\right\rceil\right)\frac{NU}{p} \\ &= \left[\left(1+\frac{1}{g}\right)\frac{bpN}{(N-h)} + 2\left(1+\left\lceil\frac{bp\tau}{T_f}\right\rceil\right)\right]\frac{NU}{p} \end{aligned} \tag{7.5}$$

For the case that N is indivisible by p, the maximum number of nodes for a parity group is $\lceil N/p \rceil$ and the minimum transmission time of packets for a parity group is $\lfloor N/p \rfloor Q/bR_v$, so the total buffer requirement is changed to:

$$B_{t,p} = \left(1+\frac{1}{g}\right)\frac{N^2}{(N-h)}Q + 2\left(1+\left\lceil\tau\Big/\left\lfloor\frac{N}{p}\right\rfloor\frac{Q}{bR_v}\right\rceil\right)\left\lceil\frac{N}{p}\right\rceil U \tag{7.6}$$

Note that the buffer sharing technique discussed in section 4.3 from video blocks that are retrieved for local playback cannot be applied here because a node

belongs to only one of the p parity groups. Therefore buffer sharing cannot be used for all but the group that the node belongs to.

7.3 System Response Time

With reference to section 4.4, system response time is the sum of scheduling delay and prefetch delay. Scheduling delay is not affected by the multiple parity group scheme. Prefetch delay is the sum of data retrieval time and the time to receive the first parity group. The former is not changed but the latter is reduced by a factor of p by the multiple parity group scheme. Using derivations similar to (4.15) we can obtain the prefetch delay as

$$\begin{aligned} D_{p,p} &= T_g + \left(1 + \left\lceil \frac{bp\tau}{T_f} \right\rceil\right)\frac{T_f}{bp} \\ &= \left[\frac{1}{g} + \frac{1}{bp}\left(1 + \left\lceil \frac{bp\tau}{T_f} \right\rceil\right)\right] T_f \end{aligned} \tag{7.7}$$

For the case that N is indivisible by p, the minimum number of nodes for a parity group is $\lfloor N/p \rfloor$ and the minimum transmission time of packets for a parity group is $\lfloor N/p \rfloor Q/bR_v$, so the prefetch delay is changed to:

$$D_{p,p} = T_g + \left(1 + \left\lceil \tau \Big/ \left\lfloor \frac{N}{p} \right\rfloor \frac{Q}{b}\frac{1}{R_v} \right\rceil\right)\left\lfloor \frac{N}{p} \right\rfloor \frac{Q}{b}\frac{1}{R_v} \tag{7.8}$$

7.4 Redundancy Configuration

In this section, we investigate the tradeoff between system response time, buffer requirement, and redundancy level. The previous sections have shown that by dividing a cluster into multiple parity groups, we can reduce the system response time and the buffer requirement. The tradeoff is an increase in the redundancy level needed to maintain the same system MTTF.

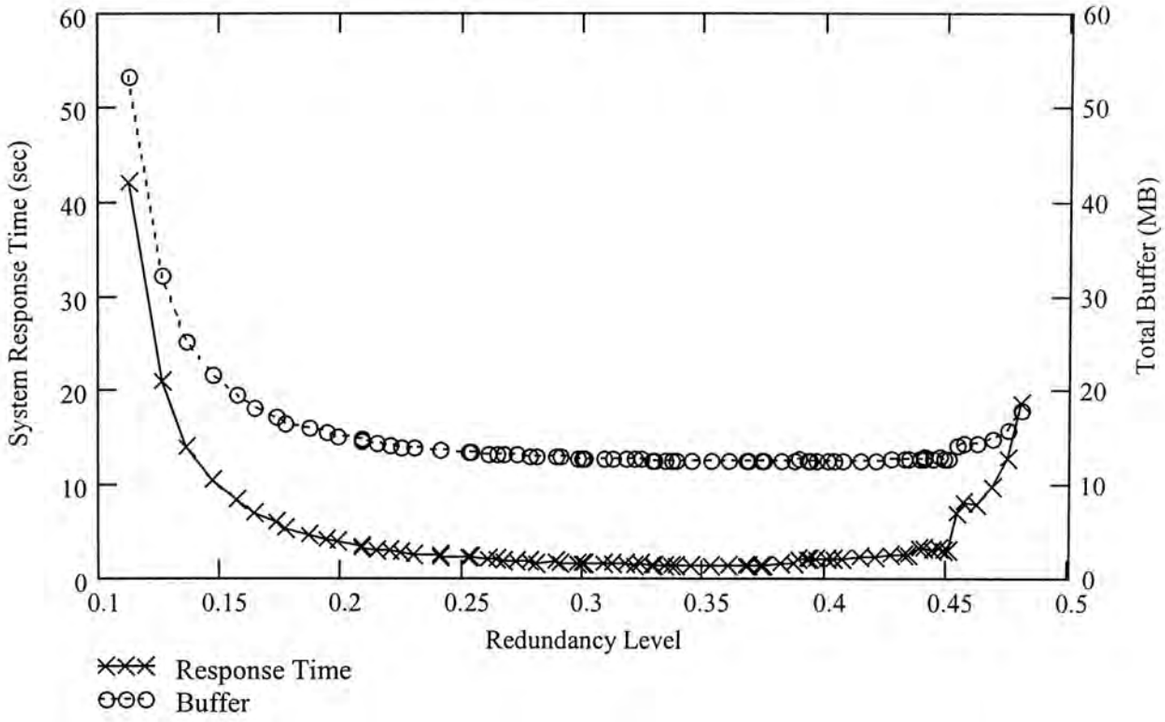

Figure 7.2. System response time and total buffer requirement versus redundancy level under the multiples parity groups scheme (N=1500, System MTTF=10,000 hours).

Figure 7.2 plots the inter-play between these three system parameters for a 1,500-node cluster achieving a system MTTF of 10,000 hours. As expected, both system response time and buffer requirement decrease with more redundancy (i.e. more parity groups). The improvement levels off for redundancy level over 0.2.

Surprisingly, configuring the system with too much redundancy (i.e. too many parity groups) can be counter-productive, and results in increased system response time and buffer requirement. This observation is explained by the fact that at high redundancy levels, the playable video data transmitted in each round decreases. Thus to compensate, the macro round length will need to be shortened, resulting in reduced disk efficiency. Finally, to guarantee sufficient disk throughput, the GSS scheduler will need to be configured with fewer groups, resulting in larger rounds with more

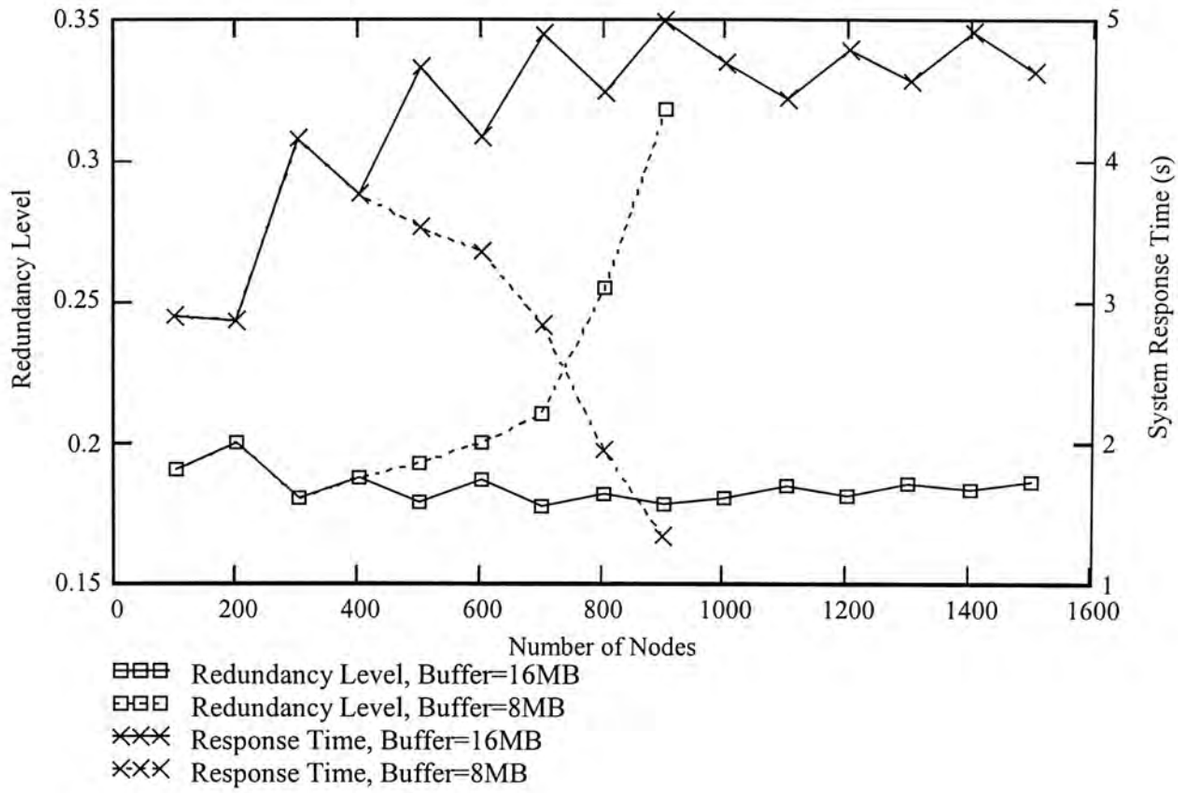

Figure 7.3. System response time and redundancy level versus cluster size under buffer constraint (System MTTF=10,000 hours).

video blocks retrieved in each round. Consequently, this increases the retrieval buffer requirement (cf. Equation (7.3)) and lengthens the scheduling delay.

7.5 Scalability

The multiple parity group scheme provides an effective tool to combat the primary system bottleneck – the system response time. However, whenever one bottleneck is removed, another one is revealed. The question is where the new bottleneck is and what is the effect on the system's scalability.

To investigate this issue, we plot in Figure 7.3 the redundancy level required and the system response time versus cluster size up to 1,500 nodes. Two sets of results are presented, one for a buffer constraint of 8MB and the other 16MB. The number of parity groups is configured such that the system response time stays within 5 seconds and the system MTTF stays above 10,000 hours.

We first consider the case of 16MB buffer constraint. The results show that as the cluster size increases, the system response time also increases, but quickly levels off and stays below the 5-seconds constraint. The redundancy level, on the other hand, stays relatively constant as the increased cluster size results in improved redundancy efficiency (cf. Figure 5.4) that compensates for the increased redundancy overhead incurred by the multiple parity group scheme.

Next we consider the case with only 8 MB buffer requirement. For small cluster sizes (below 400) the results are similar but beyond that the response time decreases and the redundancy level increases dramatically. This surprising result is due to the small buffer size available and the fact that the transmission buffer requirement increases linearly with cluster size (cf. Equation (7.3)). Therefore when the cluster size is increased, the system is forced to configure with increasingly more parity groups to reduce the receiver buffer requirement to stay within the 8MB buffer constraint. As a result, the redundancy overhead is sharply increased and the system response system is sharply reduced. Eventually, even the multiple parity group scheme cannot compensate for the increasing transmission buffer size and the system scalability is limited to 900 nodes. Hence the system scalability bottleneck has shifted to the buffer requirement

Nevertheless, comparing Figure 7.3 with Figure 6.2, it is clear that the multiple parity group scheme has already extended the system scalability significantly. Moreover, today's clients such as STB and PCs can support far more than 8MB buffer, thus further extending the system scalability. In the extreme case where the scalability ceiling is reached, one can always divide the system into multiple independent clusters to further extend the system scale.

Chapter 8

CONCLUSIONS AND FUTURE WORKS

In this thesis, we proposed a server-less architecture for building scalable, reliable, and cost-effective VoD systems without the need for dedicated video servers. We presented designs for the data placement policy, the retrieval and transmission scheduler, and a scalable fault tolerance mechanism based on multiple parity groups. Based on these designs, we derived the performance models to quantify the storage requirement, network bandwidth requirement, buffer requirement, and the system response time. We further modeled the system reliability using a continuous-time Markov chain model to obtain the system MTTF and introduced a multiple parity group scheme to extend the scalability of the architecture. Results show that we can build server-less VoD clusters as large as 1,500 nodes with system response time less than five seconds, system MTTF over 10,000 hours, redundancy overhead less than 20%, and per-node buffer requirement less than 16MB. These figures compare favorably to even dedicated high-end video servers despite that fact that the server-less architecture does not need a dedicated server at all. Moreover, we can further scale up the system scale by forming additional autonomous clusters and there is no inherent limit to the scalability of the system.

This study is a first step taken to establish several fundamental properties of a server-less VoD architecture. There are many other possibilities and challenges in the design of a server-less VoD system. For example, this study assumes that clusters are autonomous and independent. Relaxing this assumption to allow clusters to communicate with one another will open many more design choices for data placement policy, fault tolerant mechanism, and so on. Within a cluster, one can also use different striping policies for different video titles. For example, it may be desirable to use smaller striping group for more popular video titles to increase their availability and also at the same time to reduce response time. Again, more investigations are needed to quantify the performance gain and the tradeoffs.

APPENDIX

A. Derivation of the Artificial Admission Delay

Let the client node request a new video session at time t during the group $k = \lfloor tg/T_f \rfloor$. Then due to the clock jitter (τ) and the maximum transmission time (δ) of the request packets, the working group at other nodes after receiving the packet can range from $\lfloor (t+\delta)g/T_f \rfloor$ to $\lfloor (t+\tau+(N-1)\delta)g/T_f \rfloor$. To avoid uneven group assignment among nodes which can result in increased receiver buffer requirement, the new session should always be assigned a group that has not started in any nodes in the cluster. This can be done by assigning the new session to the group

$$\begin{aligned} k_n &= \left\lfloor \frac{(t+\tau+(N-1)\delta)g}{T_f} \right\rfloor + 1 \\ &\le \left\lfloor \frac{tg}{T_f} \right\rfloor + \left\lceil \frac{(\tau+(N-1)\delta)g}{T_f} \right\rceil + 1 \\ &= k + \left\lceil \frac{(\tau+(N-1)\delta)g}{T_f} \right\rceil + 1 \end{aligned} \tag{A.1}$$

This is equivalent to delaying the admission by an artificial delay of $\Omega = \left\lceil \frac{(\tau+(N-1)\delta)g}{T_f} \right\rceil + 1$ groups.

B. Derivation of the Receiver Buffer Requirement

Recall that macro round duration is the time required for each node sends b transmission blocks of U bytes ($Q=bU$) to the receiver. From (3.2),

$$\frac{T_f}{b}=\frac{NQ}{bR_v}=\frac{NU}{R_v} \tag{A.2}$$

The receiver buffer is divided into two parts. The first one is pre-fetch buffer, the video playback starts only when this pre-fetch buffer is filled up. This buffer is used for preventing data underflow while the second part of receiver buffer is used for preventing data overflow. Consider the buffer is arranged in groups of N blocks of U bytes in size, suppose group zero consists of transmission block zero to (N–1), group one consists of block N to ($2N$–1) and so on. We consider the buffer as a circular buffer of total l groups, in which y groups are pre-fetch buffer and z groups are for preventing overflow (ie. $l=y+z$). Let the transmission starts at time t_0, and $F(j)$ be the time for group j to be completely filled up,

$$\begin{aligned} t_0+(j+1)\frac{NU}{R_v} \le F(j) \le t_0+(j+1)\frac{NU}{R_v}+\tau \\ t_0+(j+1)\frac{T_f}{b} \le F(j) \le t_0+(j+1)\frac{T_f}{b}+\tau \end{aligned} \tag{A.3}$$

The playback begins after the pre-fetch buffer is filled up (ie. $F(y-1)$), and the playback time $P(j)$ for group j is,

$$\begin{aligned} P(j) &= F(y-1)+j\frac{NU}{R_v} \\ &= F(y-1)+j\frac{T_f}{b} \end{aligned} \tag{A.4}$$

To avoid buffer underflow, we must ensure the requirement that the playback time of group j buffer to be later than the fill-up time,

$$
\begin{aligned}
F(j) &\le P(j) \\
F(j) &\le F(y-1) + j\frac{T_f}{b} \\
t_0 + (j+1)\frac{T_f}{b} + \tau &\le t_0 + y\frac{T_f}{b} + j\frac{T_f}{b}
\end{aligned}
\tag{A.5}
$$

Rearranging, we can obtain the minimum requirement on pre-fetch buffer,

$$
y \ge 1 + \frac{b\tau}{T_f} \tag{A.6}
$$

Under the situation of circular buffer, we need to ensure the playback time of group j buffer to be earlier than the fill-up time of group (j+l–2) to avoid buffer overflow, because we must have at least one free group available for receiving data at any time.

$$
\begin{aligned}
P(j) &\le F(j+l-2) \\
F(y-1) + j\frac{T_f}{b} &\le F(j+l-2) \\
t_0 + y\frac{T_f}{b} + j\frac{T_f}{b} + \tau &\le t_0 + (j+l-1)\frac{T_f}{b} \\
\tau + \frac{T_f}{b} &\le (l-y)\frac{T_f}{b}
\end{aligned}
\tag{A.7}
$$

Substituting $z = (l-y)$, we can obtain the minimum requirement to prevent buffer overflow:

$$
z \ge 1 + \frac{b\tau}{T_f} \tag{A.8}
$$

Combining the two results, we obtain the receiver buffer requirement from

$$
B_r = 2\left(1 + \left\lceil \frac{b\tau}{T_f} \right\rceil\right) NU \tag{A.9}
$$

BIBLIOGRAPHY

[1] G. On, M. Zink, M. Liepert, C. Griwodz, J. B. Schmitt, R. Steinmetz, "Replication for a distributed multimedia system" *Proceedings of the Eighth International Conference on Parallel and Distributed Systems*, 2001, pp.37–42.

[2] D. N. Serpanos; L. Georgiadis; T. Bouloutas, "MMPacking: a load and storage balancing algorithm for distributed multimedia servers", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 1, Feb. 1998, pp.13–17.

[3] J. Y. B. Lee, "Parallel Video Servers: A Tutorial", *IEEE Multimedia*, vol. 5, no. 2, Apr.-Jun. 1998, pp.20–28.

[4] J. Y. B. Lee, "Concurrent push-A scheduling algorithm for push-based parallel video servers", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 9, no. 3, Apr. 1999, pp.467–477.

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

[6] M. Y. Wu and W. Shu, "Scheduling for large-scale parallel video servers", *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computing*, Oct. 1996, pp.126–133.

[7] J. Y. B. Lee, "Supporting server-level fault tolerance in concurrent-push-based parallel video servers", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 1, Jan. 2001, pp.25–39.

[8] D. N. Serpanos, A. Bouloutas, "Centralized versus distributed multimedia servers", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, no. 8, Dec. 2000, pp.1438–1449.

[9] R. Tewari, D. M. Dias, R. Mukherjee, and H. M. Vin, "High availability in clustered multimedia servers", *Proceedings of the 12th International Conference on Data Engineering*, 1996, pp.645–654.

[10] J. Gafsi, E. W. Biersack, "Modeling and performance comparison of reliability strategies for distributed video servers", *IEEE Transactions on Parallel and Distributed Systems*, vol.11, no. 4, Apr. 2000, pp.412–430.

[11] Napster. http://www.napster.com.

[12] Gnutella. http://gnutella.wego.com.

[13] M. Parameswaran, A. Susarla, and A. B. Whinston, "P2P networking: an information sharing alternative", *Computer*, vol. 34, no. 7, Jul. 2001, pp.31–38.

[14] G. Fox, "Peer-to-peer networks", *Computing in Science & Engineering*, vol. 3, no. 3, May-Jun. 2001, pp.75–77.

[15] M. Satyanarayannan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", *IEEE Transactions on Computers*, vol. 39, no. 4, Apr. 1990, pp. 447–459.

[16] W. J. Bolosky, J. R. Douceur, D. Ely, M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs", *Proceedings of the international conference on Measurement and Modeling of Computer Systems*, 2000, pp. 34–43.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage", *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 190–201.

[18] H. Weatherspoon, J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison", *Proceedings for the 1st International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[19] A. L. N. Reddy, J. C. Wyllie, "I/O issues in a multimedia system", *Computer*, vol. 27, no. 3, Mar. 1994, pp.69–74.

[20] P. S. Yu, M. S. Chen, and D. D. Kandlur, "Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management", *ACM Multimedia Systems*, vol. 1, no. 3, 1993, pp.99–109.

[21] D. L. Mills, "Internet time synchronization: The network time protocol", *IEEE Transaction on Communications*, vol. 39, no. 10, Oct. 1991, pp.1482–1493.

[22] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Englewood Cliffs, NJ: Prentice-Hall, 1995, pp.227–234.

[23] A. J. McAuley, "Reliable Broadband Communication Using a Burst Erasure Correcting Code", *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, Sept. 1990, pp. 297–306.

[24] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols", *ACM Computer Communication Review*, vol. 27, no. 2, Apr. 1997, pp24–36.

[25] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, L. A. Rowe, "Multimedia storage servers: a tutorial", *Computer*, vol. 28, no. 5, May 1995, pp.40–49.

[26] G. Ganger and J. Schindler, "Database of Validated Disk Parameters for DiskSim", http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html.