Thèse

présentée pour obtenir le grade de docteur

de l'École Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

JAMEL GAFSI

DESIGN AND PERFORMANCE OF LARGE SCALE VIDEO SERVERS

Soutenue le 17 Novembre 1999 devant le Jury composé de:

Président	Prof. Jacques Labetoulle (Eurecom)
Rapporteurs	Prof. Wolfgang Effelsberg (Universität Mannheim) Prof. Monique Becker (INT)
Examinateurs	Dr. Isabelle Demeure (ENST) Dr. Zhen Liu (INRIA)
Directeur de Thèse	Prof. Ernst W. Biersack (Eurecom)

École Nationale Supérieure des Télécommunications

Dedication

To my mother and the memory of my father.

I hope to be now what my father taught me to be.

Acknowledgments

First and foremost I would like to thank my thesis advisor Professor Ernst Biersack for his support and his guidance over the last three years. He never stopped guiding, encouraging, and supporting me, especially during the very hard time I had after my father died. I am indebted to him for provoking my interest to video on demand servers and for teaching me how to write papers and to give talks. His continuous and thorough reviews improved tremendously the technical quality and the presentation of the work I have published.

I am grateful to Prof. Jacques Labetoulle, Prof. Wolfgang Effelsberg, Prof. Monique Becker, Dr. Isabelle Demeure, and Dr. Zhen Liu for agreeing to serve on my dissertation committee and for the time they took to read and comment my work.

Special thanks go to Didier Loisel and David Tremouillac who were the main forces behind the administration of the hardware and material we have required. I owe special thanks to Ulrich Walther, Jan Exner, and Thomas Mancini for their contributions in the implementation of our video server prototype. I also thank Mme Evelyne Biersack, Emmanuelle Chassagnoux, and Karim Maouche for reading and correcting the french summary of this dissertation.

I have immensely benefited from many stimulating discussions, technical and otherwise, with many colleagues at Eurecom. I will fondly remember the countless hours I spent discussing with my colleagues and friends at Eurecom: Jörg Nonnenmacher, Jakob Hummes, Christian Blum, Lassaad Gannoun, Raymond Knopp and later Arnaud Legout, Morsy Cheikhrouhou, Matthias Jung, Pablo Rodriguez, Neda Nikaein, Sergio Loureiro, Arnd Kohrs, Pierre Conti, and Guiseppe Montalbano. Special thanks to Arnaud Legout for his permanent help.

I am especially indebted to three fabulous friends, Jörg Nonnenmacher, Jakob Hummes, and Morsy Cheikhrouhou. I thank them for their help on innumerable occasions, their advice on technical matters and their perspectives in life in general. I have learned a lot from them.

My life at the Côte d’Azur has been enriched by several people I have gotten to know over the past three years. The long road towards a doctoral degree can be fraught with frustration and pain. Instead, the company and support of these friends have made this journey mostly rewarding and fun.

Finally, I would like to thank my family for sharing the ups and downs of my student life and for putting up with very long years of separation.

Résumé

Les applications multimédia, qui commencent à apparaître, vont devenir omniprésentes dans quelques années. Un exemple de ces applications est la vidéo à la demande. La mise en oeuvre de celle-ci nécessite la conception de nouveaux systèmes de stockage et de livraison appelés serveurs vidéo. La conception de ces derniers doit tenir compte de la nature de l'information vidéo qui est très volumineuse, gourmande en bande passante et impose des contraintes en matière de délais de livraison.

La conception d'un serveur vidéo représente plusieurs défis: celui-ci doit servir une grande population de clients simultanément. En outre, il doit être robuste au facteur d'échelle (*scalable*) et doit être aussi économiquement rentable. Finalement, son architecture doit tolérer les pannes de ses composantes afin de garantir un service ininterrompu. L'objectif de cette thèse est de concevoir et étudier la performance d'un serveur vidéo qui réalise ces défis.

Cette thèse identifie, propose et compare plusieurs algorithmes qui interviennent dans les différentes phases de conception d'un serveur vidéo. Elle étudie en particulier l'architecture du serveur vidéo, le placement et la distribution des données vidéo et la fiabilité du serveur vidéo. Nous proposons un algorithme de répartition des données sur plusieurs disques et noeuds du serveur vidéo, appelé Mean Grained Striping, et nous le comparons avec les algorithmes de répartition des données que nous avons identifiés en matière du débit du serveur (nombre maximum des clients admis simultanément), du besoin en buffer et du temps de latence initial pour un nouveau client. Nous avons considéré le cas d'un serveur vidéo non-tolérant aux pannes et celui d'un serveur vidéo tolérant aux pannes. Nos résultats montrent surtout que l'algorithme de répartition des données et celui qui assure la fiabilité du serveur vidéo sont *interdépendants* et le choix de l'un doit être pris en combinaison avec le choix de l'autre. En outre, nous comparons plusieurs algorithmes de fiabilité du serveur vidéo en fait de la performance et du coût du serveur. Les résultats prouvent que pour un serveur vidéo, la technique de fiabilité fondée sur la simple réplication des données est moins coûteuse que celle qui est fondée sur la technique de parité. Afin d'évaluer quantitativement la fiabilité du serveur vidéo pour les différentes méthodes de fiabilité, nous modélisons la fiabilité à l'aide des chaînes Markoviennes. L'évaluation de ces modèles montre que l'algorithme de fiabilité Grouped One-to-One, que nous avons proposé, assure la fiabilité la plus importante en dépit d'un coût par flux relativement élevé. Nos résultats indiquent aussi que diviser le serveur vidéo en petits groupes indépendants aboutit au meilleur compromis entre une fiabilité élevée et un coût par flux bas. Dans le cas d'un serveur vidéo qui utilise la technique de réplication des données, nous proposons une nouvelle méthode de placement de la réplication, appelée *ARPS* (Adjacent Replica Placement Scheme). Celle-ci place les données originales directement à côté des données répliquées de façon à éliminer les temps de recherche supplémentaire quand le serveur vidéo opère dans le mode de défaillance. Nous montrons que ARPS améliore le débit du serveur vidéo de $60-90\%$ par rapport aux méthodes classiques de placement de la réplication. Finalement, nous implémentons un prototype de serveur vidéo qui reflète les décisions que nous avons prises durant la phase de conception. Le prototype implémente un nouvel algorithme distribué d'ordonnancement et d'extraction des données. En outre, nos résultats expérimentaux montrent que le prototype du serveur vidéo est robuste au facteur d'échelle en matière du nombre de noeuds contenus dans le serveur vidéo.

Abstract

The concept of Video On Demand (VOD) has received lot of attention during the past few years. To make VOD possible, one needs to address the issues of digital video storage and retrieval using a video server.

The design of video servers mainly poses four challenges: (i) the video server must achieve a high throughput to serve a large number of customers concurrently, (ii) it must be scalable, (iii) it must provide a cost effective architecture, and (iv) it must be reliable that is guaranteeing an uninterrupted service even when operating with some component failures. The design and the performance evaluation of a video server that addresses these four challenges is the focus of this dissertation.

This dissertation describes, proposes, and compares several algorithms and techniques that concern the video server architecture, data layout, data striping that is the technique of distributing video data on the video server, and server reliability. We introduce a generic data striping scheme, called the Mean Grained Striping algorithm, and compare different striping algorithms in terms of server throughput, buffer requirements and start-up latency for a non fault-tolerant as well as a fault-tolerant video server. Our results demonstrate that the choice of the striping scheme must be made in combination with the choice of the reliability scheme. Moreover, we compare several reliability schemes in terms of server performance and cost. We find that mirroring-based reliability is more cost effective than parity-based reliability for video servers. In order to quantitatively evaluate server reliability for the different reliability schemes that we have identified, we perform reliability modeling based on Markov chains. The identification of the different reliability schemes is based on the technique used to achieve reliability and on the distribution granularity of redundant data. We propose a mirroring-based organization, called the Grouped One-to-One scheme and show that it outperforms all other mirroring- and parity-based schemes in terms of video server reliability at the expense of a slightly higher per stream cost. The results further indicate that dividing the video server into small independent groups of disks achieves the best trade-off between high server reliability and low per stream cost. For mirroring-based reliability, we propose a novel replica placement scheme that stores replicated data adjacently to original data. This scheme outperforms existing replica placement schemes in terms the overall server throughput. Finally, we instantiate most of the design decisions in a prototype implementation. The prototype addresses the main goals that were followed during the design phases and implements a distributed stream scheduling and retrieval at the server side and a Java platform-independent client. Furthermore, the experimental results show that the prototype is scalable in terms of the number of nodes that are contained on the video server.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The incredible progress in computing and communication technologies over the past years has made it feasible and economically viable to provide new services such as video-on-demand, interactive television, online news, distance learning, and multimedia messaging. All of these services involve storing, accessing, delivering, and processing multimedia information (video and audio). The realization of such services requires the development of multimedia storage servers, i.e. video servers, that are capable of transmitting multimedia information to thousands of users. These users will be permitted to retrieve multimedia objects from the video server for real-time playback.

There are two fundamental characteristics of digital video and audio that make the design of video servers significantly different from the one of traditional file systems and storage servers. The first characteristic concerns the *real-time requirements* of video (audio) data. Indeed, in contrast to conventional data (e.g. text, binary files) that do not have rate requirements, digital video (audio) consists of a sequence of video frames (audio samples) that convey meaning only when presented continuously in time. The second characteristic of video data are *large storage* and *high data transfer rate* requirements. In fact, even in their compressed form, video data remain voluminous (a 100 minutes MPEG2 coded video at the rate of $2\mathrm{Mbit/sec}$ is about $1.2\mathrm{GBit}$ large), where the term MPEG refers to Motion Picture Expert Group that has standardized many audio and video compression formats. Consequently, the storage and transfer rate requirements for a video server are *huge*, since the latter is expected to store thousands of videos and to serve many thousands of users concurrently. Given these requirements, a video server must provide (i) massive amounts of space and bandwidth to store and display video data and (ii) efficient mechanisms for storing, retrieving, and manipulating video data in large quantities and at high speeds.

Magnetic disks have established themselves as the most suitable storage devices for video servers since they have the desirable features to support video storage and retrieval. Indeed, magnetic disks allow fast random access, high data transfer rates, and have a high storage capacity at moderate prices. Given that a single magnetic disk is not capable to provide the storage and bandwidth requirements of a large video server, a video server consists typically of multiple magnetic disks organized as a disk array [Patterson 88]. Since a disk array is typically connected to a single machine, the number of its disks is limited by the capacity of the respective machine. Therefore, making the video server scalable in terms of both, the storage capacity and the throughput achieved, leads to a distributed architecture that contains multiple machines, also called nodes, where each of them behaves as a single disk array.

For a video server, the distributed solution is essential, given the large volume of data to be stored and the global throughput to provide. However, a distributed video server that is made of multiple disk arrays poses various and sometimes conflicting challenges. These challenges mainly include providing high server throughput, achieving good load balancing between the multiple server nodes, designing appropriate data storage and retrieval algorithms such as distributed stream scheduling and admission control, ensuring guaranteed and uninterrupted video data delivery even when some of the server components fail, and finally keeping server cost as low as possible. These challenges must be addressed together since the different issues are interdependent. Hence the need to a complete *design* that best compromises the challenges cited above while meating the server storage, retrieval, and real-time requirements. The design and performance evaluation of a distributed video server is the focus of this dissertation.

A key point of our work is that different design aspects of a video server are related and each of these aspects can have an impact on the others. Therefore, the design of a video server must simultaneously consider the different aspects together. We address throughout this dissertation *three* central design aspects that are *data layout*, *data striping*, and *server reliability*.

Data layout determines the placement order in which video data is stored on server disks. We consider two types of data layout. The first type is *intra-disk layout* that organizes the physical data placement order within the surface of a single disk, whereas the second type is *inter-disk layout* that addresses data layout between the different disks of the video server. Data layout is addressed in different chapters of this dissertation.

Before defining data striping, let us discuss the following scenario. Given is a video object to store on the video server. If this video object is entirely stored on a single disk, the number of concurrent accesses to that video object is limited by the transfer rate capacity of that disk. A solution to overcome this limitation is to copy the video object several times on different disks. However, this solution results in wasted storage space and thus in high server costs. Another drawback of entirely storing a video object on a single disk are load-imbalances that might occur between server disks. In fact, a disk that contains a popular video object may experience a *hot spot*, whereas other disks may remain underutilized or even idle. An effective solution to these problems is to *partition* the video object and *scatter* it across multiple disks, which is referred to as **data striping**. There is no standard and unique definition of data striping in the literature. The following is the data striping definition we consider. *Data striping partitions a given video object into blocks that are stored on a set of disks. Successive blocks of the video object are stored on different disks following, for instance, a round robin manner*. There are mainly three advantages of data striping for video servers: Given a very popular video object that is stored across *all* server disks, the whole server capacity can be used to serve the clients that are all requesting that popular video object, which allows for high throughput without the need to duplicate the video object several times. Further, since blocks of each video object are contained on all server disks, perfect load balancing is achieved independently of which video objects are requested. Finally, when new disks are added to the video server, a simple redistribution of the blocks of each video object allows to exploit the additional server capacity and thus making the video server scalable. Different data striping algorithms differ (i) in the size of the blocks to store on disks and (ii) in the number of disks that are involved to retrieve video data during a certain time interval. The analysis and comparison of several data striping algorithms with respect to video server performance and quality of service is the focus of chapters 3 and 4.

The third design aspect we concentrate on in this thesis is video server reliability or video server fault-tolerance[1]. In order to ensure high quality of service and uninterrupted service, the development of new reliability concepts that match the characteristics and requirements of a large and distributed video server is necessary. In fact, even though modern disk drives have a mean time to failure (MTTF) of many years, a sufficiently large collection of disk drives can experience frequent failures. For instance, for a disk MTTF of $300,000 \ \mathrm{hours}$ and given that disk life times are exponentially distributed [GIBS 90], a video server with 300 disks has a mean time between failures (MTBF) of only $\frac{300,000}{300} = 1,000 \ \mathrm{hours}$, which is only $42 \ \mathrm{days}$. However, the support of reliability within a video server introduces a storage and bandwidth overhead that may affect server performance and costs. In chapters 4, 5, and 6, we study video server reliability with respect to different design goals such as high server performance, low server cost, high server reliability, and also good load balancing. We only consider server reliability schemes that add redundant information to original information in order to make the video server fault-tolerant. We focus on server reliability from two different points of view: the reliability technique used and the distribution granularity of redundant data. We analyze the video server behavior during normal operation mode, where all server components are operating, and during failure mode, where one or some of the server components fail. In the context of reliable video servers, we also address the management and scheduling of data retrieval for a distributed video server during both, normal operation mode and failure mode. Appendix B derives appropriate procedures for data scheduling and retrieval during these two modes.

[1]The terms reliability and fault tolerance are used interchangeably throughout this thesis.

1.1 Dissertation Outline

The rest of this dissertation is organized as follows.

Chapter 2 provides a design overview for a video server. We first motivate the choice of magnetic disks as unique video server storage devices. We then highlight the video server environment and design phases and derive server performance and quality of service metrics. We finally articulate the primary design decisions that will be the basis during the rest of this dissertation.

Chapter 3 studies data striping for a non-fault tolerant video server. We classify several striping algorithms. We then present the Mean Grained striping algorithm, a generic algorithm that covers the most relevant striping schemes. We compare the retained data striping algorithms in terms of the overall server throughput, buffer requirement, and start-up latency for new incoming client requests.

Chapter 4 studies video server fault-tolerance and its effect on server performance and quality of service based on the striping algorithm used. We compare striping algorithms regarding server throughput, buffer requirement, and start-up latency when the video server operates in failure mode and make explicit the relationship between the striping algorithm and the reliability technique used.

Chapter 5 first classifies several server reliability schemes depending on the technique used and the distribution granularity of redundant data. We then perform reliability modeling for the different schemes identified based on Continuous Time Markov Chains. Thereby, we split the discussion to the two cases of independent disk failures and dependent component failures. We propose a novel mirroring scheme, called Grouped One-to-One mirroring and show that it achieves highest video server reliability. We compare the retained reliability schemes with respect to server reliability and per stream cost and determine, for schemes that divide the video server into independent groups, the group size that best trades-off server reliability and per stream cost.

Chapter 6 presents a novel replica placement scheme for mirroring-based video servers. We first classify interleaved declustering schemes for mirroring. We then propose ARPS (Adjacent Replica Placement Scheme) that stores original data adjacent to replicated data and compare it with the classical interleaved declustering schemes in terms the overall server throughput.

Chapter 7 instantiates the different video server design aspects in a prototype implementation. We describe the different components of the distributed video server prototype and present the algorithms we have implemented for distributed stream scheduling, failure detection, and failure recovery. We present experimental results to demonstrate the scalability of our video server prototype.

Finally, chapter 8 summarizes our results and outlines directions for future work.

Chapter 2

Video Server Design Issues

The main challenges in the design of a video server are (i) to satisfy the video server requirements regarding storage, retrieval and real-time constraints and (ii) to allow for a scalable and cost effective video server architecture [Srivastava 97]. The aim of this chapter is first to give an overview of the video server design. This includes the devices considered to store and retrieve video data, the overall video server environment, the video server design phases, and finally the video server performance and quality of service criteria. The second part of this chapter focuses on the primary design decisions we make, which are the basis of all the work carried out in this dissertation.

2.1 Design Overview

Due to the storage, bandwidth, and real-time requirements of a video server, its design fundamentally differs from the one of classical file systems. Further, the video server design must result in a scalable and fault-tolerant architecture at a low cost. As we will see throughout this dissertation, the different issues of the video server design are interdependent and must be therefore addressed together. To point out the most relevant design aspects for a video server, we give next a design overview, where we first motivate the choice of magnetic disks as storage devices for a video server and discuss the main characteristics of a magnetic disk drive. Subsequently, we focus on the video server environment and identify the video server design phases. The last part of this section introduces the different server performance and quality of service criteria that we will use later to evaluate and compare various algorithms that can be involved in the different server design phases.

2.1.1 Video Server Storage Devices

The first decision to make when designing a video server is the choice of the storage device that meets best the space and I/O bandwidth capacity requirements as well as the real-time constraints of the video server [Chung 96]. Common storage devices are tapes, optical disks, CD-ROM, DVD, magnetic disks, and main memory (RAM). Out of all these storage devices, only magnetic disks have the desirable features to support storage and retrieval for video servers [Lu 96, Chervenak 94, REDD 96]. In fact, magnetic disks provide (i) fast and random access unlike tapes and optical disks, (ii) high data transfer rates in contrast to tapes, optical disks, CD-ROM, or DVD, (iii) high storage capacity unlike CD-ROM and DVD, and finally (iv) moderate cost unlike main memory. Moreover, the capacity and speed of magnetic disks has improved tremendously over the last 20 years. The storage capacity of a magnetic disk increases at a rate of 25% per year and it exceeds today $18 \mathrm{GByte}$ [Grochowski 97b]. Further, magnetic disks experience a steady transfer rate improvement (40% per year). Currently, the new generation of magnetic disks, e.g. the IBM Ultra 36XP SCSI disk family, has a transfer rate of about $40 \mathrm{MByte/sec}$ [Grochowski 97c]. Another encouraging factor for choosing magnetic disks as storage devices is their price that continuously decreases at a rate of 40% per year [Grochowski 97a]. Today (1999), the price of of magnetic disks is about $\$0.2$ per $1 \ \mathrm{MByte}$ [Grochowski 97a, Mr.X]. However, the average access time of magnetic disks has improved over the last 15 years *only* at a rate of 5% per year. This average access time is today about $10 \mathrm{ms}$.

Note that magnetic disk costs were relatively high a few years ago, which motivated some researchers to propose storage hierarchies in order to construct a cost-effective video server [Doganata 96, Shastri 98, KIENZ 95, WILK 96]. All of these works suggest to store only a small set of frequently requested video objects on magnetic disks. The large set of less frequently requested video objects are stored on tapes or optical disks that were much cheaper than magnetic disks. Today, magnetic disks provide large storage capacity at a low price, which allows to use them as unique storage device for a video server and thus to fully exploit their high throughput and low latency advantages compared to other storage devices such as tapes and optical disks. A deep knowledge of the internals of magnetic disk[1] drives is needed to understand well the storage and retrieval requirements and thus to make the right design decisions for a video server.

Characteristics of Magnetic Disk Drives

Regarding mechanical components of disk drives and their characteristics, we are based on [Mr.X] and also on the work done in [WILK 94]. The SCSI (Small Computer System Interface)

[1] In the remainder of this book, the term disk will refer to magnetic disk when it is used alone

standard presents thereby the most popular magnetic disk drive. As shown in Figure 2.1, a disk drive typically contains a set of **surfaces** or platters that rotate in lockstep on a central spindle. Each surface has an associated disk head responsible for reading/writing data. Unfortunately, the disk drive has a single read data channel and therefore only one disk head is active at a time. A surface is set up to store data in a series of concentric circles, called **tracks**. One track is made of multiple storage units called **sectors**, where each sector commonly holds $512 \mathrm{bytes}$ of data. Tracks belonging to different surfaces and having the same distance to the central spindle build together a **cylinder**. As an example of today's disks, the seagate Barracuda ST118273W disk drive contains 20 surfaces; $7,500$ cylinders; and $150,000$ tracks.

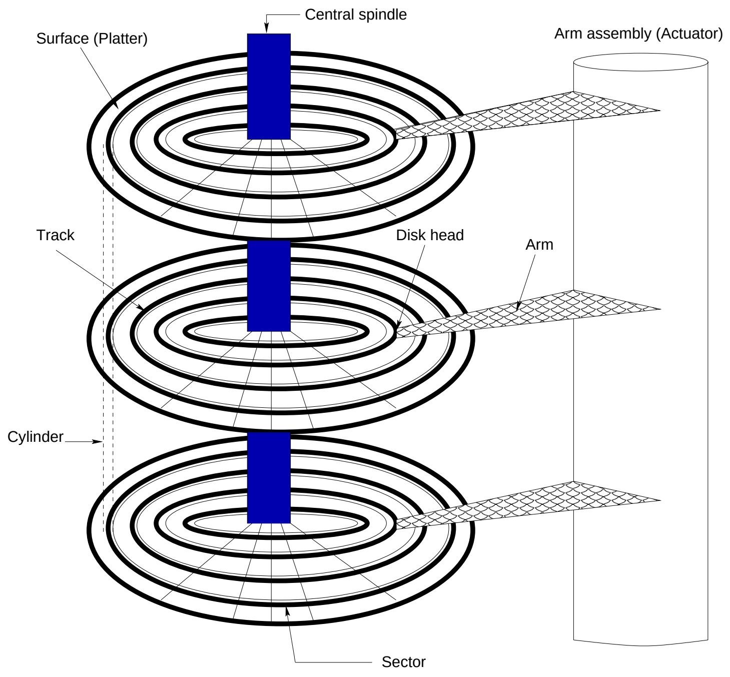


Figure 2.1: Magnetic Disk Characteristics.

In order to access data stored on a given track, the disk head must perform two operations. First, the disk head must be positioned over the correct track. The time that elapses to reposition the disk head from the current track to the target track is known as **seek time**. The maximum value of the seek time t_{seek} is the time that the disk head spends to move from the outer track to the inner track or inversely. More precisely, seek time as studied in [WILK 94, WORT 95] is mainly composed of four phases: (i) a *speedup* phase, which is the acceleration phase of the arm, (ii) a *coast* phase (only for long seeks), where the arm moves at its maximum velocity, (iii) a *slowdown* phase, which is the phase to rest close to the desired track, and finally (iv) a *settle* phase, where the disk controller adjusts the head to access the desired location. Note that the duration t_{stl} of the settle phase is independent of the distance traveled and is about $t_{stl} = 3 \text{ ms}$. However, the durations of the speedup phase (t_{speed}), the coast phase (t_{coast}), and the slowdown phase ($t_{slowdown}$) mainly depend on the distance traveled. The seek time t_{seek} takes then the following form: $t_{seek} = t_{speed} + t_{coast} + t_{slowdown} + t_{stl}$.

Once the disk head is positioned at the target track after performing a seek operation, the disk surface must rotate such that the desired data is moved under the disk head. The time needed for this operation is termed **rotational latency**. In other terms, rotational latency is the time the disk arm spends inside one cylinder to reposition itself on the beginning of the data to be read. The maximum value of the rotational latency t_{rot} is directly given by the rotation speed of the spindle. The most common spindle speeds are $4,500 \ \mathrm{rpm}$, $5,400 \ \mathrm{rpm}$, $7,200 \ \mathrm{rpm}$, and more recently $10,000 \ \mathrm{rpm}$. For a spindle speed of $7,200 \ \mathrm{rpm}$, the maximum value of the rotational latency $t_{rot} = 8.33\mathrm{ms}$.

During seek time and rotational latency, the disk is not performing any read operation. The percentage of the disk utilization depends therefore on these two values. High seek time and rotational latency negatively affect the disk utilization. Given that the video server must implement scheduling and admission control policies to guarantee continuous real-time data delivery, the values of the seek time and the rotational latency must be based on the worst case disk head movements. We will see in later chapters that the seek time and the rotational latency at the magnetic disk level have a big impact on the design choices we have made concerning the data striping algorithm, the reliability scheme as well as the data layout and placement policy. For more details about magnetic disk drive characteristics, we refer to [WILK 94, WORT 95, Hennessy 90, Mr.X].

2.1.2 Video Server Environment and Server Design Phases

The primary goal of a video server is to achieve high performance at low cost. Hence the motivation to use *standard hardware* for the video server. Besides the price advantage, a video server that is based on standard hardware can continuously profit from the technological advances of these components. However, using (standard) cheap hardware requires to shift the video server complexity to the application and therefore to realize the whole intelligence of the server in software. This requires a careful design that must respect all video server requirements together and achieve the expected performance and quality of service goals.

In order to set the appropriate context of a video server, let us consider the environment shown in Figure 2.2. Here, the video server is based on multiple magnetic disks building a disk array. In this scenario, multiple clients request playback of stored videos from the video server. The video server retrieves data stored on its disks (1), temporarily stores the data in its buffer (2) until the data is transmitted onto the network interface (3) and through the network to the client machines (4). Each client receives its requested video stream and performs video playback simultaneously.

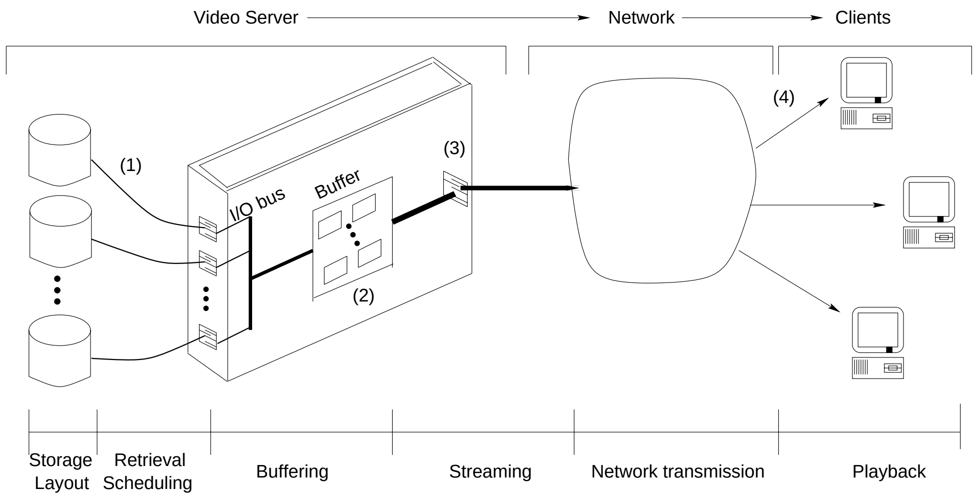


Figure 2.2: The video server environment.

The previous scenario leads to identify the following video server design aspects. *Data storage* determines the way a given video object is stored on the video server. *Data placement* and *data layout* indicate the physical placement of different parts of a video object on disks. For data layout, we make the difference between *intra-disk layout* and *inter-disk layout*. The first term organizes the data placement within a single disk, whereas the second term organizes the data placement between different disks of the video server. *Data retrieval* defines for each stream the frequency of retrieving its data from server. *Stream scheduling* gives the order in which multiple streams are served from a single disk. Since the disk transfer rate is higher than a single stream playback rate, multiple streams can be served from a single disk. However, in order to avoid starvation at the client side and thus guarantee continuous playback, *admission control* must be integrated in the video server. The admission control policy decides for a new incoming stream, whether it can be accepted. Because of the speed mismatch between data retrieval from disk and video playback at the client side, main memory (*buffer*) is needed to temporarily store retrieved data. The amount of buffer needed is a function of the number of streams served and on the amount of data read for each stream. *Streaming and network transmission* build together the data delivery phase: Data stored temporarily in the buffer must be sent to the client via the network to arrive in time. Finally, *video playback* is performed at the client side. We will focus in more details on these design aspects in section 2.2.

Figure 2.2 also emphasizes the possible bottlenecks of a video server environment. These can be (i) the disk transfer rate (also called disk I/O bandwidth), (ii) the I/O bus capacity at the SCSI interface, (iii) the buffer space, (iv) the transfer rate at the network interface, and (v) the client capacity (bandwidth and processing). The potential bottlenecks that are related to the video server capacity are (i), (ii), and (iii). In order to efficiently exploit the total capacity of the video server, the I/O bus capacity must be higher than the sum over all disks I/O bandwidth of the disk array. Once this condition holds, only two bottlenecks remain: the disk I/O bandwidth and the buffer space.

2.1.3 Video Server Performance and Quality of Service

The performance criteria considered throughout this dissertation are *server throughput*, *buffer requirement*, and *server cost*. Server throughput is defined as the maximum number of streams the video server can admit concurrently. Buffer requirement is a decisive performance measure for a video server since it can affect the server cost. Indeed, whereas main memory cost experiences a steady decrease as is the case for SCSI disk cost, main memory remains relatively expensive ($1 \ \mathrm{MByte}$ RAM cost about $\$13$ against $\$0.2$ for $1 \ \mathrm{MByte}$ of hard disk) and represents a main cost factor of the video server. The third performance metric is server cost. More precisely, we consider the cost per stream that is the overall server cost divided by the server throughput. We only consider hard disk and memory costs to calculate the per stream cost.

Besides high performance and low cost, the video server must satisfy quality of service criteria. The most decisive quality of service metric is *video server reliability* that is to ensure continuous and uninterrupted service even in the presence of some component failures. Nowadays, magnetic disk drives are fairly reliable and the mean time to failure MTTF of a single disk drive is very high (up-to $500,000$ hours for the Seagate Barracuda or Cheetah disk families [Mr.X]). However, given that a video server is made of hundreds or even thousands of disks, the probability that one or some disk failures occur becomes very high. In the context of classical file systems, reliability (also referred to as fault tolerance) was addressed in [Patterson 88], where six schemes based on RAID (Redundant Array of Inexpensive Disks) were introduced and specified. These schemes have in common that they all use redundant data to reconstruct lost data due to disk failures. Redundant data may be based on simple replication mechanisms or on parity decoding as we will show later when discussing video server reliability. However, RAID schemes as presented in [Patterson 88] are based on hardware, which makes them not adapted to real-time systems such as a video server. Hence the need for reliability mechanisms at the application level to make the video server fault-tolerant. A large part of this dissertation addresses video server reliability, see chapters 4, 5, and 6.

Another quality of service metric is the *start-up latency* that is the interval from the point in time at which the client has sent its request for a stream to the point in time at which the client receives the first frame of the requested video. Note that the calculation of the start-up latency does not include network delays. The last quality of service criterion is *load-balancing*. The even distribution of the server load among all its components is very important when designing a video server, since load imbalances may cause a dramatic reduction of video server performance.

2.2 Design Decisions

We determine next the video format we will consider for storage and retrieval. Digital video can be encoded either using constant bit rate (CBR) or variable bit rate (VBR) compression algorithms. A CBR stream has a constant consumption rate with variable quality, whereas a VBR stream has a variable consumption rate with constant quality. We will only consider CBR streams in our video server design study.

A large part of this dissertation studies two central design issues of a video server. The first issue is how to organize and store hundreds of video objects over multiple disks such that thousands of clients can concurrently access these objects, whereas the second issue is how to ensure an uninterrupted and reliable service for all clients at any point in time. Before we analyze and discuss in details these two issues, we introduce in the following the decisions we make for the other design aspects of a video server such as the overall server architecture, the scheduling, admission control, retrieval, and buffering policies considered.

The Server Array: A Scalable Architecture

We consider a disk array-based video server architecture as we have described in the previous section. A decisive metric of a video server architecture is its scalability in terms of both, the server throughput, which determines the maximum number of streams that can be serviced concurrently, and the storage capacity, which determines the number of video objects that can be stored. However, an increasing demand for bandwidth or/and storage capacity cannot be met by simply adding new disks within the disk array. Indeed, given that the disk array is connected to a single machine (node), the number of disks that are contained in this disk array is limited to the capacity of this machine. Limiting factors can be the SCSI I/O bandwidth, the system bus, the CPU, or the buffer space. Consequently, ensuring scalability for a video server consists in adding new nodes, where each node behaves in turn as a single disk array. As a result, the video server is made of multiple server nodes or multiple disk arrays. A video server that consists of multiple nodes can be designed in different ways.

One way is to build a server out of autonomous and independent nodes, where a video object is totally contained on a single node. the disadvantages of this architecture are its limited throughput and load imbalances. Indeed, the number of clients that are consuming the same very popular video object, which is stored on a single server node, is limited to the capacity of this node. Hence the limited throughput for this alternative. Further, autonomous nodes have load balancing problems, since nodes that store popular video objects can become hot spots, whereas other nodes may remain idle. Eventually, the hot spot node will be overloaded and unable to accept requests for service while other servers are not fully utilized. The only escape from this situation is to simply duplicate popular videos on more than one server node, which results in a waste of precious storage volume. Additionally, the popularity of the video objects is not always known in advance and may change over time, which requires a very complex monitoring system that redistributes the storage of different video objects whenever their popularities change.

Another way to build the video server is to configure the server nodes into the called **server array** [BERN 96b]. The server array is made of multiple server nodes (disk arrays), where each node does not store entire video objects. Instead, each video object is cut into several parts and the different parts are *distributed* over possibly all nodes of the server array. The distribution of a video object over all server nodes, which is referred to as data striping, allows for (i) perfect load balancing independently of which video objects are requested and (ii) high server throughput, since the whole server capacity can be used to service the clients that are all consuming the same very popular video. The server array scales naturally by adding new server nodes. The server array architecture is illustrated in Figure 2.3.

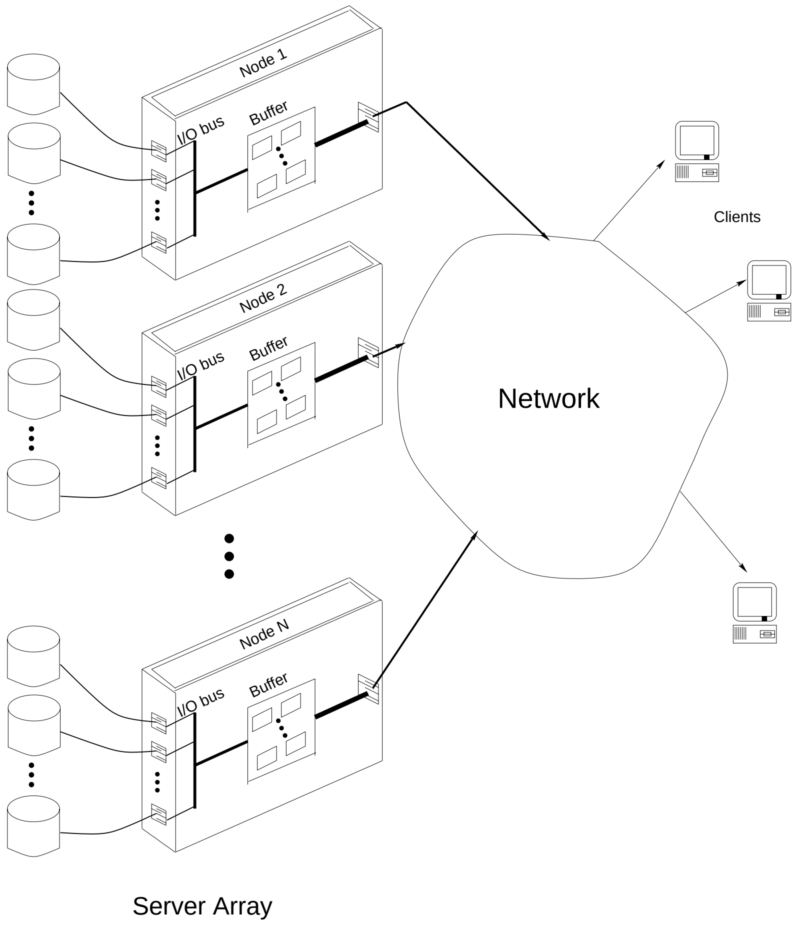


Figure 2.3: The server array architecture.

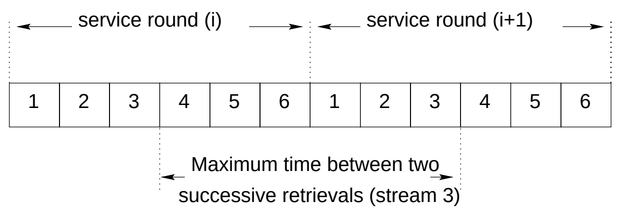
Round-Based Data Retrieval

As already mentioned, the playback of a stream must proceed at its real-time rate. Thus, the server must retrieve data from disks at a rate that avoids starvation at the client side. Hence the need to a data retrieval technique that ensures real-time data retrieval without risking data starvation. A naive retrieval technique that ensures continuous playback and thus avoids data starvation is to prefetch the whole video object from the server and buffer it prior to initiating the playback. however, such a technique requires a huge amount of buffer and results in a very high start-up latency that may not be tolerable from user. Consequently, the challenging task is to prevent starvation when retrieving data while at the same time keeping buffer requirement and start-up latency as small as possible. Given that a single disk transfer rate is significantly higher than the playback rate of a single stream, each disk is able to serve multiple streams at a given point in time. However, the video server must ensure that the continuous playback requirements of all admitted streams are met. Most of the literature considers data retrieval techniques that are based on rounds. Thereby, the video server serves multiple streams simultaneously by proceeding in service rounds. We call this technique **round-based data retrieval**, where the service time is divided into equal size time intervals called **service rounds**. More precisely, based on the server array architecture, a client is connected to all server nodes and is served *once* every service round. Data that is retrieved from video server during service round i is consumed during service round $i + 1$. Note that for CBR-coded videos, round-based data retrieval means that blocks that are retrieved for the same stream are constant length and the stream is serviced during each service round. However, the situation is more complicated for VBR-coded streams, where the literature makes the difference between the called constant time length and constant data length retrieval techniques (see [CHA 94, CHAN 96] for more details).

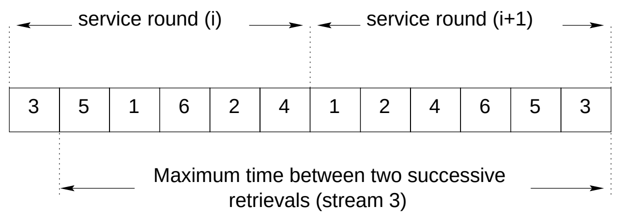
The SCAN Stream Scheduling Algorithm

Since the transfer rate of a single disk is much higher than the playback rate of a stream, each disk can serve many streams during one service round. The policy that determines the order in which blocks belonging to different streams are retrieved from a single disk, is referred to as **stream scheduling**. The stream scheduling algorithm that matches best real-time scheduling is the EDF (Earliest Deadline First) algorithm, where the block with the earliest deadline is scheduled for retrieval. EDF, however, does not order in advance the retrieval of multiple blocks. Consequently, the disk head may spend most of time in seeking the expected blocks, which results in excessive seek time and leads to a very poor disk utilization. Stream scheduling algorithms that are based on round-based data retrieval address this drawback for EDF. We discuss as follows three stream scheduling algorithms [Gemmell 95] that use round-based data retrieval.

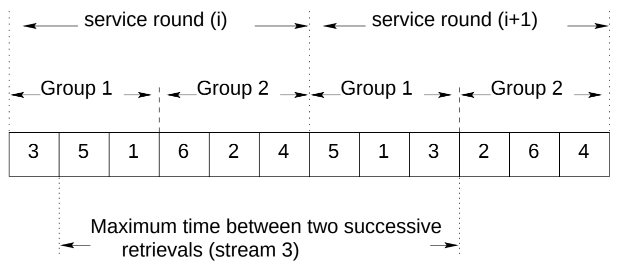
The first scheduling algorithm is Round-Robin (RR), where the order in which multiple streams



(a) The RR scheduling algorithm.



(b) The SCAN scheduling algorithm.



(c) The GSS scheduling algorithm.

Figure 2.4: The three round-based scheduling algorithms.

are serviced is kept unchanged from one service round to another as shown in Figure 2.4(a). The major drawback of RR is that it does not take into account the placement of the different blocks to be retrieved from disk and therefore the seek time can be very high, which results in poor disk utilization and low performance. To address the limitations of RR, the SCAN scheduling algorithm is used, where the disk head moves back and forth across the surface of the disk only *once* during one service round. Every block is retrieved as the disk head passes over it and thus the order in which multiple streams are serviced may change from one service to another. SCAN therefore reduces the seek overhead as compared to RR. However, SCAN requires more buffer space than RR. Indeed, since RR conserves the order in which clients are served across rounds, the retrieval times of two successive requests of a client is equal to the service round duration. Consequently, RR requires as much buffer space as needed for consumption during one service round. SCAN, however, needs twice as much buffer space, since it may happen that a client is serviced at the end of service round i and at the very beginning of the next service round $i+1$ (see Figure 2.4(b)). We see that there is a trade-off between good disk utilization (SCAN) and low buffer requirement (RR). The called Grouped Sweeping Scheme (GSS) [YU 93, CHENMS 93] was proposed to address this trade-off. The GSS algorithm partitions a service round into a number of groups. Each stream is assigned to a certain group as illustrated in Figure 2.4(c). The groups are serviced in a fixed order during each service round analogously to RR. Within each group, however, SCAN is employed. Note that if there is only one group, GSS is reduced to SCAN and if there are as many groups as clients serviced, GSS becomes equivalent to RR. By optimally deriving the number of groups, the server can balance the two conflicting goals: high disk utilization and low buffer requirement. We will consider the SCAN stream scheduling algorithm for the remainder of this thesis.

Deterministic Admission Control

Given the real-time requirements of each stream, the video server must perform admission control to decide whether a new incoming client can be admitted without violating the performance and deadline requirements of the other clients that are already being serviced. The literature makes the difference between two ways to perform admission control: the first way is based on statistical admission control and the second way is based on deterministic admission control.

With statistical admission control, deadlines are guaranteed to be satisfied with a certain probability that is to guarantee for instance that 90% of deadlines will be met during a certain service round. Statistical admission control is especially worthwhile for a video server configuration, where the admitted clients are *frequently* sending requests to the server to get pieces of the required videos. This scenario, which is called *pull mode*, uses statistical analysis to determine whether a new incoming client can be admitted and to calculate for it the data loss probability. A Ph.D. thesis has studied the performance and reliability of multi-disk multimedia servers that are based on the *pull mode* [Kaddeche 98] . In this work, a multi-disk multimedia was studied in terms of its quality of service such as the system response time, the transmission quality expressed in block loss probability, and the maximal workload that can server the system. After validating the multi-disk architecture, multimedia server fault-tolerance was studied, where both RAID1 and RAID5 were considered. This thesis further presents a set of models and simulations to evaluate multimedia servers that operate in a pull mode.

The second way to perform admission control is the deterministic one. Deterministic admission control can not be easily used for the *pull mode* explained above. However, it is adapted to the called *push mode*, where during normal play back operation, the video server sends periodically video data to each admitted client. In other terms, based on the push model and upon admission, a client becomes *passive* and receives periodically video data without requesting them, unless it uses VCR functions. Deterministic admission control ensures that *all* deadlines are guaranteed to be satisfied. To do so, the worst case situation must be considered when admitting new incoming requests.

We adopt a deterministic admission control criterion at the magnetic disk level. This criterion determines whether the disk can admit a new client, which depends on the disk characteristics (disk transfer rate, seek time and rotational latency), the video playback rate, the size of the blocks to be retrieved, and finally the scheduling algorithm used, which is SCAN in our case.

The admission control policy is simply based on the following constraint: *The service round duration must not exceed the playback duration of the already buffered information.* As a result, the following equation holds:

$$\tau \leq \frac{b}{r_p}$$

where τ denotes the service round duration and b and r_p denote the block size and the playback rate of the stream respectively. Thus, the admission control policy allows to determine the maximum number of streams Q_d that can be admitted from a single disk. In fact, this maximum is reached when $\tau = \frac{b}{r_p}$. Let t_{seek} and t_{rot} denote the maximum seek time and the maximum rotational latency of the disk. Since the SCAN scheduling algorithm is employed, the disk head moves at most twice across the whole disk surface and the worst case seek overhead during a service round is twice as much as the maximum seek time t_{seek} of the disk head. However, each stream requires in the worst case the maximum rotational latency t_{rot}. Finally, assuming that all blocks that are retrieved on a disk have the same size b and all streams have the same playback rate r_p, the admission control criterion for a single disk is [DENG 96]:

$$Q_d \cdot (\frac{b}{r_d} + t_{rot}) + 2 \cdot t_{seek} = \frac{b}{r_p} = \tau$$

Consequently, Q_d is given by Eq. 2.1.

$$Q_d = \frac{\frac{b}{r_p} - 2 \cdot t_{seek}}{\frac{b}{r_d} + t_{rot}} \tag{2.1}$$

Note that for the server array that contains multiple nodes, each containing in turn multiple disks, the admission control criterion becomes dependent on the number and size of the blocks that are retrieved for a single stream during one service round. The data striping algorithm used *clearly* and *unambiguously* determines the way a stream is served during a service round. Therefore, the formula of the admission control criterion, and thus the server throughput, differs for different striping algorithms (see chapter 3).

Double Buffering

Because of the asynchronous nature of data retrieval from disk, the video server needs to temporarily store retrieved video data in a buffer before it sends them to the respective client via the network. Let us consider a single stream served from a single disk, where a block of size b is retrieved from disk during each service round. In order to avoid buffer overflow, the amount of buffer needed for this stream is twice the value b as shown in Figure 2.2. This buffer requirement of size $2 \cdot b$ is referred to as double buffering in most of the literature. However, if we consider the case of multiple streams served by a video server that contains multiple disks, the situation changes. In [GB 97a, GABI 97], we have studied two buffer management strategies for distributed video servers. The first strategy is the called *dedicated* buffer management that allocates for each stream its worst case buffer space, which is $2 \cdot b$, independently of the other streams. Instead, the called *shared* buffer management assigns to all streams a single buffer and considers the worst case scenario over all streams together. We have calculated the buffer requirement for these two strategies and for the three scheduling algorithms RR, SCAN, and GSS. The results of this work demonstrate that for all scheduling algorithms considered, shared buffer management reduces the buffer requirements up-to 50% as compared to dedicated buffer management. Given that each stream is assigned a dynamically changing portion of a common buffer for shared buffer management, this strategy is, however, more complicated than dedicated buffer management. In the next chapters, we will explicitly indicate, which buffer management we consider.

2.3 Summary

In this chapter, we first gave an overview of the video server design. We advocated the choice of magnetic disk drives for video data storage and examined their main characteristics. We then introduced the video server environment and presented the performance and quality of service metrics for a video server. The second part of this chapter presented the different design decisions we made, which mainly include (i) the server array as video server architecture, (ii) round-based data retrieval, (iii) the SCAN stream scheduling algorithm, and (iv) a deterministic admission control criterion. The next chapters adopt these design decisions when we study other video server design aspects such as data layout, data striping, and server reliability.

Chapter 3

Data Striping

We consider the server array architecture as presented in the previous chapter. To efficiently utilize the server array, a video object is distributed over multiple disks of the server. A scheme that partitions a video object into blocks and distributes these blocks on different disks in a well defined order is called **data striping** [SaGa 86]. Data striping has been addressed previously in the literature in the context of classical file systems, see e.g. [Patterson 88, Chen 90, LEE 92, KATZ 92, CHEN 93, CHEN 94a, CLGK 94]. However, the data striping techniques proposed for classical file servers are not directly applicable for video servers due to (i) the real time requirements, (ii) the periodic and sequential nature, and (iii) the large data rate requirements of video data. Hence the need for data striping techniques that are adapted to video server requirements. Classification, analysis, and comparison of data striping algorithms for video servers are subject of this chapter.

The rest of this chapter is organized as follows. In section 3.1, we discuss data striping techniques for video servers and describe the Mean Grained Striping algorithm we propose. Section 3.2 presents related work. In section 3.3, we compare the three data striping algorithms retained in terms of server throughput, buffer requirement, and start-up latency for new incoming requests. Finally, section 4.4 summarizes our results.

3.1 Data Striping Techniques for Video Servers

In contrast to simply storing a whole video object on a single disk, striping that video object among multiple disks allows for high throughput and good load balancing. As the video server grows and thus the number of its disks increases, the striping algorithm used becomes decisive in terms of server performance and costs. In this section, we first give the storage and retrieval parameters and their respective definitions that are repeatedly used throughout this thesis. Then, we classify several striping techniques depending on their striping granularity and introduce a generic algorithm, called **Mean Grained Striping**, that covers the most relevant striping policies.

3.1.1 Storage and Retrieval Parameters

We need to introduce the relevant terms used for the storage and retrieval of video data to and from a video server:

- **Video Object**: A sequence of frames such as a movie or a video clip.
- **Video Segment**: A partition of a video object. Each of the video segments contains a contiguous part of the whole video object. The video segment consists of multiple frames.
- **Physical Disk Block**: The smallest amount of consecutive data fetched from disk in one I/O access. b_d is the size of the physical disk block.
- **Striping Unit**: The amount of contiguous logical data belonging to a single video segment and stored on a single disk. b_{su} denotes the size of a striping unit and is a multiple of the physical block size b_d (block interleaved and not bit/byte-interleaved striping).
- **Disk Retrieval Block**: The amount of data retrieved for one stream (client) from a single disk during a service round. Generally, a disk retrieval block may contain multiple striping units belonging to different video segments and thus it does not necessarily contain contiguous data. b_{dr} is the size of the disk retrieval block and is a multiple of both, b_{su} and b_d.
- **Retrieval Unit**: The whole amount of data retrieved for one stream during one service round. The retrieval unit is a multiple of the disk retrieval block and can be read either from a single disk of the server or from a group of disks of the server. The retrieval unit contains one or more video segments. b_{ru} is the size of the retrieval unit and is a multiple of b_{dr}.

Based on the definitions above, we introduce in Table 3.1 the storage and retrieval parameters considered. Note that during a single service round, up to Q_d disk retrieval blocks belonging to Q_d different video streams (clients) are read from a single disk and put in the server buffer. The value of the service round duration τ depends on the retrieval unit size b_{ru} of each stream. In fact, b_{ru} is the amount of data consumed at the client side during exactly one service round. Thus, to avoid hiccup-free display and to prevent for buffer overflow, the display time of the retrieval unit must be equal to the value $\tau = \frac{b_{ru}}{r_p}$ (see the admission control criterion in section 2.2).

Term	Definition
D	The total number of disks in the video server
N	The total number of server nodes in the video server
D_n	The number of disks that belong to a server node ($D_n = \frac{D}{N}$)
r_d	The transmission rate of a single disk in Mbit/sec
r_p	The playback rate of a video object in Mbit/sec
Q_d	The maximum number of clients that can be simultaneously served from a *single* disk
Q	The maximum number of clients that can be simultaneously served from the video server
τ	The service round duration

Table 3.1: Storage and retrieval parameters

3.1.2 Striping Techniques

Data striping is the technique of partitioning of a video object into video segments that are in turn distributed across a set of disks in a predefined order. When we have multiple disks, we need to decide over how many disks to distribute (1) the whole video object *(video object striping)* and (2) each individual video segment *(segment striping)*. In the following, we introduce three video object and three video segment striping techniques. The combination of video object striping and video segment striping unambiguously defines how the data of a video object is stored on the disks. We show in Figures 3.1 to 3.6 for a video server with 6 disks simple examples of the striping techniques we will discuss below.

Video Object Striping (v_s)

If we consider an entire video object, we can identify:

Video-Single-Striping (v_{ss}): A naive storage method is to store each video object on *a single* disk. Let us assume three video objects $VO1$, $VO2$, and $VO3$ that are stored respectively on three disks d_1, d_2, and d_3, as depicted in Figure 3.1. v_{ss} stores each video object on a single disk and thus it suffers from load imbalance and lack of robustness in case of disk failures: If $VO1$ is very popular and $VO2$, and $VO3$ are not required at all, the disks d_2 and d_3 will be underutilized, whereas the bandwidth of d_1 is not sufficient to serve all clients that simultaneously request $VO1$. Additionally, a disk failure (e.g. d_1) results in loss of a whole video object ($VO1$).

Video-Narrow-Striping (v_{ns}): The distribution of a video object across multiple disks gives a higher throughput, especially when many clients require the same video object. When we

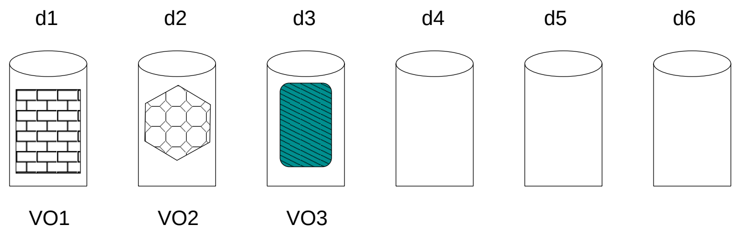


Figure 3.1: Video-Single-Striping (v_{ss})

distribute a video object only across a subset of disks of the server, we deal with the v_{ns} mechanism. Figure 3.2 shows the storage of 2 video objects $VO1$ and $VO2$, where $VO1$ is distributed over disks d_1, d_2, and d_3, whereas video object $VO2$ is distributed over disks d_4, d_5, and d_6.

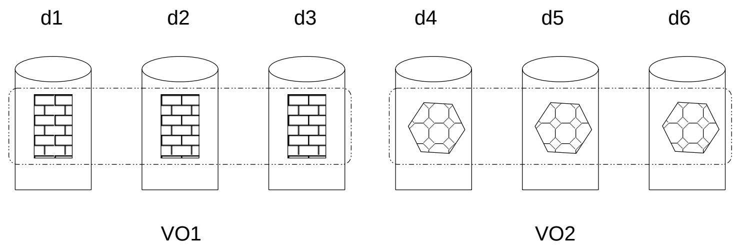


Figure 3.2: Video-Narrow-Striping (v_{ns})

Video-Wide-Striping (v_{ws}): This technique distributes each video object over *all* existing disks of the server. An example showing the storage layout of 2 video objects $VO1$ and $VO2$ is depicted in Figure 3.3.

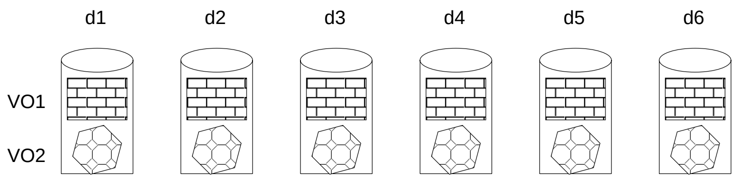


Figure 3.3: Video-Wide-Striping(v_{ws})

Video-Segment-Striping (s_s)

If we consider a single segment, we can identify:

Segment-Single-Striping (s_{ss}): A video segment is entirely stored on a single disk (Figure 3.4). In this case, the disk retrieval block, the retrieval unit, and the video segment are all equivalent. s_{ss} can be combined with all three video object striping policies v_{ss}, v_{ns}, and v_{ws}.

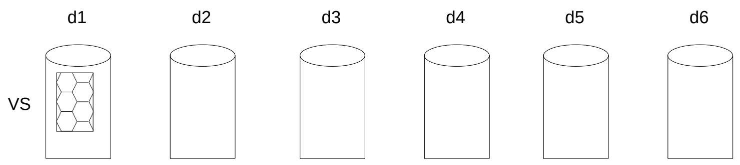


Figure 3.4: Segment-Single-Striping (s_{ss})

Segment-Narrow-Striping (s_{ns}): This means that every video segment is stored on a subset of the disks (Figure 3.5). s_{ns} can be combined with v_{ns} and v_{ws}.

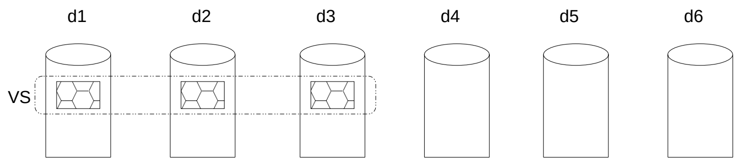


Figure 3.5: Segment-Narrow-Striping (s_{ns})

Segment-Wide-Striping (s_{ws}): In contrast to s_{ns}, s_{ws} partitions each video segment into many sub-segments as disks there are on the server. Thus, all available disks of the server are involved to store *a single video segment* (see Figure 3.6). s_{ws} can be only combined with v_{ws}.

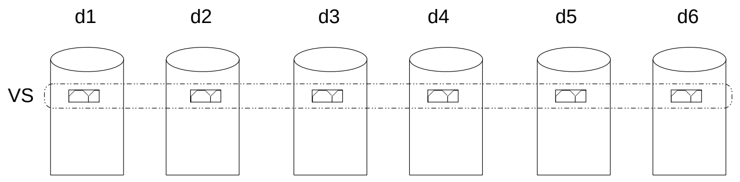


Figure 3.6: Segment-Wide-Striping (s_{ws})

Video Object vs. Video Segment Striping

We now study the different combinations of video object and segment striping and describe the storage and retrieval parameters of a video server for each possible combination. Let U_{vo} denote the **video object size** and U_{vs} the **video segment size**. Assume that all video segments are equal size and the size of each stored video object is a multiple of the size of a video segment. To simplify the discussion, we define the following two functions as:

1. $\nu(v_s)$: Returns the type of v_s. We have distinguished between three different strategies: v_{ss}, v_{ns}, and v_{ws}. Thus: $\nu(v_s) \in \{v_{ss}, v_{ns}, v_{ws}\}$

2. $\sigma(s_s)$: Returns the value of the s_s. We have also distinguished between three different strategies: s_{ss}, s_{ns}, and s_{ws}. Thus: $\sigma(s_s) \in \{s_{ss}, s_{ns}, s_{ws}\}$

A particular striping strategy is described with a tuple $(\nu(v_s), \sigma(s_s))$ that indicates the unique combination of v_s and s_s. As we have already seen, some combinations are not possible. In the following, we describe every possible combination in terms of the relationship that may exist between the following system parameters: the physical disk block size b_d, the striping unit size b_{su}, the disk retrieval block size b_{dr}, the retrieval unit size b_{ru}, the video segment size U_{vs}, and the video object size U_{vo}:

- (v_{ss}, s_{ss}) : Each video object is stored on a single disk and therefore also all its video segments. We have $U_{vo} = b_{su} \gg b_{ru} = U_{vs} = b_{dr} \gg b_d$
- (v_{ns}, s_{ss}): Each video object is stored on a set of disks. However, each video segment is stored on a single disk. We have $U_{vo} > b_{su} = b_{ru} = U_{vs} = b_{dr} \gg b_d$
- (v_{ws}, s_{ss}): Each video object is stored on all available disks and each video segment is stored on a single disk. We have $U_{vo} \gg b_{su} = b_{ru} = U_{vs} = b_{dr} \gg b_d$
- (v_{ns}, s_{ns}): Each video object is stored on a set of disks. Each video segment of a video object is stored on the same set of disks or on a subset of it. We have $U_{vo} > b_{ru} > b_{dr} > b_{su} \gg b_d$
- (v_{ws}, s_{ns}): Each video object is stored on all available disks. A video segment of a given video object is stored on only a set of disks. We have $U_{vo} \gg b_{ru} > b_{dr} > b_{su} \gg b_d$
- (v_{ws}, s_{ws}): Each video object is stored on all available disks. Each video segment is also distributed on all disks. We have $U_{vo} \gg b_{ru} \gg b_{dr} \gg b_{su} \geq b_d$

3.1.3 MGS: A Generic Data Striping Algorithm

After having classified various striping techniques, we now limit our further discussion to some of these techniques for deep and detailed study. To do this, we have looked at the possible striping algorithms and realized that:

1. Video *object* striping should be video wide striping (v_{ws}) where a video object is distributed over *all* disks of the video server: v_{ws} achieves a good *load-balancing* independently from the video objects requested and offers *highest throughput* for popular video objects as compared to video narrow striping v_{ns} and video single striping v_{ss}.
2. For video *segment* striping, three approaches are possible:

- s_{ws} distributes *each segment over all disks* and therefore achieves perfect load balancing. However, as we will see, the buffer requirement grows proportionally to the number of disks.
- s_{ss} stores the whole segment on a *single* disk, which can result in a load imbalance between disks and high start-up latency for new client requests. Given that a large segment is entirely stored on a single disk, the number of disk accesses for one stream is small, resulting in small seek overhead.
- s_{ns} distributes a video segment over a *sub-set* of all disks and can be considered as a compromise between s_{ws} and s_{ss}.

Now combining our choice for video wide striping v_{ws} with the three possible segment striping techniques s_{ws}, s_{ss}, and s_{ns}, we are then left with three possible striping techniques that we will further investigate:

- (v_{ws}, s_{ws}), which will be referred to as **FGS** or **Fine Grained Striping** [OZDE 96b].
- (v_{ws}, s_{ss}), which will be referred to as **CGS** or **Coarse Grained Striping** [OZDE 96b, BEBA 97].
- (v_{ws}, s_{ns}), which will be referred to as **MGS** or **Mean Grained Striping** [GABI 98c].

Retrieval Groups

To define where a video segment should be stored, we introduce the notion of **retrieval group**. A retrieval group can comprise *one*, or *multiple* disks. Each retrieval unit is read from *one* retrieval group during a service round. A single disk of the server belongs to *exactly one* retrieval group. The **retrieval group size** is the number of disks belonging to a retrieval group and determines the striping granularity of a video segment.

The following parameters are needed to model a video server that is based on several retrieval groups:

- C: Number of retrieval groups in the server.
- D_c: Retrieval group size: D_c indicates how many disks the retrieval unit will be simultaneously read from during one service round.
- Q_c: Maximum number of clients that can be simultaneously served by a retrieval group.

One can vary the retrieval group size D_c within a video server. We assume that all retrieval groups have the same size (D_c). Thus, D is a multiple of C and: $D_c = \frac{D}{C} \quad \forall c \in [1..C]$

Varying the retrieval group size allows us to cover the three striping algorithms CGS, FGS, and MGS:

- $Dc = 1$: The retrieval unit is entirely stored on *one* disk (CGS) and is equal to the disk retrieval block that, in turn, equals the video segment.
- $D_c = D$: The retrieval unit is distributed over *all* disks of the server (FGS) and is D times larger than the disk retrieval block.
- $1 < D_c < D$: The retrieval unit is distributed over a *set* of disks D_c of the server (MGS) and is D_c times larger than the disk retrieval block.

An MGS Striped Video Server

The assignment of disks to nodes and to retrieval groups can be carried out as follows: Let d_k denote a disk with $k \in [0, ..., (D-1)]$. If we define $n = (k \text{ div } D_n)+1$ and $l = (k \bmod D_n)+1$, then disk d_k is the l-th disk of node n and belongs to retrieval group c with $c = (((l-1) \cdot N + n)\ div\ D_c) + 1$.

Figure 3.7 shows an example of an MGS striped video server, where each retrieval group contains one disk from each server node as is the case for the *orthogonal RAID* configuration that was used in [GIBS 90]. A retrieval group g contains the disks d_k with $k = i \cdot D_n + g - 1$ and $i \in [0, ..., (N-1)]$.

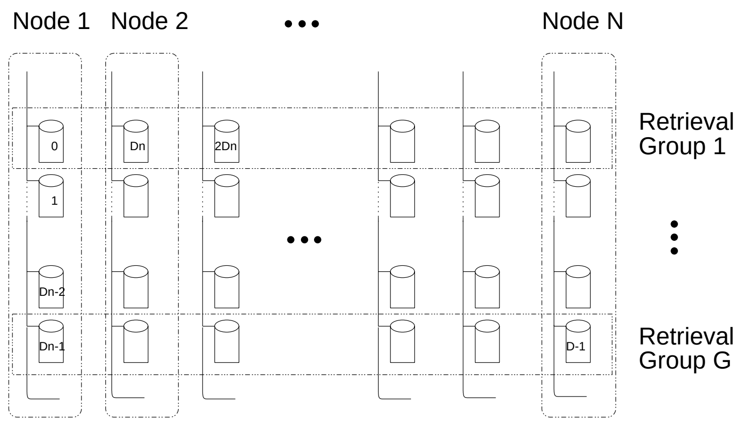


Figure 3.7: Retrieval Groups

Parameters Below are the parameters that specify a video object stored on a video server:

- V is the number of video objects to store in the video server. VO_i denotes the video object i ($i \in [1..V]$.

- $U_{vo}(i)$ denotes the size of the video object i and $VS_{i,j}$ is the j^{th} video segment of the i-th video object.
- $U_{vs}(i,j)$ represents the size of $VS_{i,j}$. For the sake of simplicity, we assume that for a given video object $U_{vo}(i)$: $\exists n, \exists U_{vs} | \; U_{vo}(i) = n \cdot U_{vs}$.

We present in the following the MGS algorithm that is based on Figure 3.7. The MGS algorithm presented is a simple example of the MGS class. Depending on the striping granularity of a video segment, MGS can configure each retrieval group containing more than *one* disk from each server node, or consisting of disks from a sub-set and not all server nodes.

MGS Algorithm

```
for all video objects $VO_i$ ($i \in [1..V]$) {
  Partition $VO_i$ into video segments $VS_{i,j}$
  with:  $j \in \left[1..\left(\frac{U_{vo}(i)}{U_{vs}}\right)\right]$.

  for all video segments $VS_{i,j}$ {
    Partition $VS_{i,j}$ into $D_c$ striping units $S_{i,j,k}$
      where $k \in [1..D_c]$.
    Determine the retrieval group $c$ that will contain
      $VS_{i,j}$ such that:  $c = (i+j-1) \bmod C$.
    Store the striping unit $S_{i,j,k}$ on the disk $d$
      with:  $d = c + (k-1) \cdot D_n$.
  }
}
```

The MGS algorithm presented above ensures perfect load-balancing inside a retrieval group. Disks belonging to the same retrieval group store the same amount of video data. Further, consecutive video segments $VS_{i,j}$ and $VS_{i,j+1}$ are stored across consecutive retrieval groups c and $((c+1) \mod C)$. This ensures an equal distribution of video data across all retrieval groups and distributes the storage load as fairly as possible. Additionally, the two first video segments $VS_{i,1}$ and $VS_{i+1,1}$ of two consecutive video objects VO_i and VO_{i+1} are stored across consecutive retrieval groups (i mod C) and $((i+1) \mod C)$ to better distribute new client requests over the retrieval groups.

In the following, we consider one of the retrieval groups shown in Figure 3.7 and show how one retrieval unit R_u of a video object i is stored on the different disks of the retrieval group. Figure 3.8 illustrates the storage of different striping units $S_{i,j,k}$ and video segments $VS_{i,j}$. It also shows the disk retrieval block D_{rb} to be retrieved from a single disk. Let us assume that $b_{ru} = m \cdot U_{vs}, \; m \in \{1, 2, ..\}$. A disk retrieval block contains striping units of m video segments.

By making the disk retrieval block contain *multiple* striping units, we can optimally trade off disk access overhead and main memory requirements. When $m = 1$, the striping unit and the disk retrieval block are the same size and the video segment size equals the retrieval unit size.

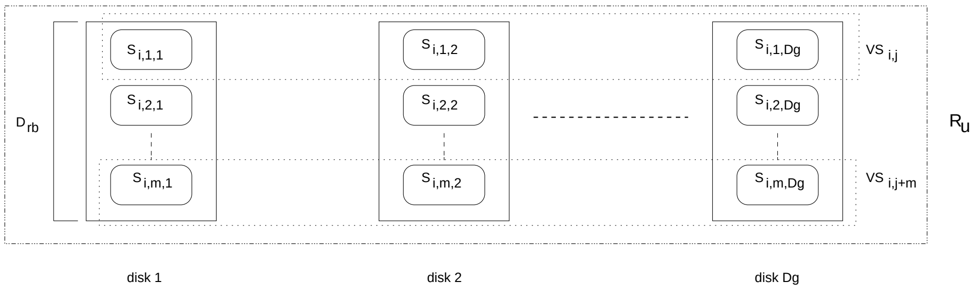


Figure 3.8: Striping of a retrieval unit over a retrieval group

Many researchers have proposed data striping schemes that are similar to the MGS algorithm proposed above, where the video server is divided into different groups. The most relevant schemes are the streaming RAID [TOB 93b], the staggered striping scheme [BEGH 94], the configuration planner [GHAN 95b]. However, none of these schemes takes into account a multiple node video server architecture as the case for our server array. Instead, the MGS algorithm considers a multiple node video server and has the originality of exploiting the orthogonality principle that was introduced in [GIBS 90]. In fact, retrieval groups are independent of each others and disks belonging to the same retrieval group are contained on different nodes, which allows to fairly distribute the server load among the different nodes and also to tolerate even a complete node failure as we will see in later chapters of this thesis.

3.2 Related Work

Various papers have investigated data striping in video servers. According to the classification of section 3.1.3, we discuss the striping algorithms proposed in the literature. In Table 3.2, we attribute to each striping technique its corresponding striping class depending on the combination of the video object and the video segment striping granularity. The symbol ”XXX” indicates combinations that are not allowed.

In [SHEN 97], a (v_{ns}, s_{ns}) striping algorithm was proposed that distributes a video object only on a set of disks. Its main disadvantage is that it does not distribute all video objects uniformly on all disks and therefore popular video objects will be replayed only from a few disks. On the other hand, the bandwidth-imbalances between disks becomes higher when the number of disks increases. Because of its restriction in terms of the number of concurrent streams requiring the same video object, we will not consider this striping algorithm in the later discussion.

$(\sigma(s_s), \nu(v_s))$	$\sigma(s_s)$=s_{ss}	$\sigma(s_s)$=s_{ns}	
$\nu(v_s)$ = v_{ss}	No striping	XXX	
$\nu(v_s)$ = v_{ns}	Shenoy/Vin [SHEN 97] (large Segments) Berson et al. [BER 94a]	Shenoy/Vin [SHEN 97] (small Segments)	
$\nu(v_s)$ = v_{ws}	Oezden et al. [OZDE 96a, OZDE 96b] Mourad [MOUR 96, Mourad 96] Tewari et al. [TEWA 96a]	Berson et al. [BEGH 94] Ghandeharizadeh et al.[GHAN 95b] Tobagi et al.[TOB 93b]	Oezden et al. [

Table 3.2: Classification of Striping Strategies

In [OZDE 96b], (v_{ws}, s_{ws}) and (v_{ws}, s_{ss}) were compared in terms of the maximum number of admitted streams (**throughput**) given a fixed amount of resources on the video server. The results show that (v_{ws}, s_{ss}) achieves a higher throughput than (v_{ws},s_{ws}). This study did not take into account either the latency overhead for every client request or fault-tolerance.

(v_{ws}, s_{ss}) was also studied in [MOUR 96], where fault tolerance is assured using a mirroring method (the doubly striped mirroring) that uniformly distributes the load of a failed disk over all remaining disks.

In [TOB 93b], streaming RAID was proposed, where a video object is stored on the server using (v_{ws},s_{ns}). The video server is divided into fixed size clusters. This work clearly emphasizes the constraints on the number of admitted streams due to the disk I/O bandwidth, available buffer and start-up latency as a function of the retrieval unit size, which is the amount of data read for a stream during a single service round. However, this work does not compare the streaming RAID scheme with the other parity schemes as we will do later in this chapter.

In [TEWA 96a], the authors use the streaming RAID approach and additionally propose two schemes to distribute parity information across all disks: the storage of a parity group can be sequential or random. The goal is to distribute the load uniformly over disks when working with or without a single disk failure. The authors do not study the performance of the server in terms of throughput, buffer, start-up latency.

In [GHAN 95b], the striping granularity was discussed and a planner was proposed to decide the cluster size. The authors proposed to split a video segment across one, some, or all disks in the server depending on the desired throughput and latency. This organization corresponds to (v_{ws}, s_{ns}). Only throughput and latency were addressed to determine the way a video segment should be striped on disks.

In [BEGH 94], the staggered striping (v_{ws}, s_{ns}) was proposed to improve the throughput for concurrent access compared with the so-called virtual data placement [1]. The staggered striping method especially allows popular video objects to be striped over all available clusters and thereby avoids replicating them many times to achieve the expected bandwidth.

We note that none of the works evaluated and compared the three retained data striping algorithms FGS, CGS, and MGS for a multiple node video server in terms of *all* of the following criteria: throughput, buffer requirement, start-up latency, and load-balancing.

3.3 Comparison of the Striping Algorithms

In this section, we compare the three retained data striping algorithms FGS, CGS, and MGS in terms of buffer requirement (section 3.5.5.2), video server throughput (section 3.3.3.2), and start-up latency (section 3.3.3.3).

3.3.1 Buffer Requirement

Since the transmission rate r_d of a single disk is much larger than the playback rate r_p of a video object, a single disk can serve multiple clients at the same time. Since for a particular client the data retrieval will be ahead of the data consumption, main memory is needed at the server side to temporarily store video data.

For a single stream, the buffer requirement varies over time since it is determined by the difference between production of video by the server and consumption by the client.

We consider the following assumptions to calculate the amount of buffer needed:

1. The total number of clients Q that can be admitted is assumed to be constant. Let Q^{FGS}, Q^{CGS} and Q^{MGS} denote the maximum number of admitted clients for respectively FGS, CGS, and MGS: we assume that $Q = Q^{FGS} = Q^{CGS} = Q^{MGS}$.

2. SCAN is the scheduling algorithm used.

3. The buffer requirement is for the case of *shared* buffer management where each video stream has been assigned a dynamically changing portion of a common buffer. As already mentioned, compared to dedicated buffer management, where each stream, *independently* of the other streams, is assigned as much buffer as it needs in the worst case, shared buffer management reduces the buffer requirement by up to 50% [GB 97a, GABI 97].

[1]The virtual data placement assumes a system consisting of d clusters and each video object is assigned to a single cluster (v_{ns}, s_{ns}).

Buffer Requirement for FGS, CGS, and MGS

Table 3.3 gives the values of the parameters C, D_c and Q_c depending on the striping algorithm used.

Parameter	FGS	MGS	CGS
C	1	$\frac{D}{D_c}$	D
D_c	D	$\frac{D}{C}$	1
Q_c	Q	$\frac{Q}{C}$	Q_d

Table 3.3: Design Parameters for FGS, CGS and MGS

When we use the parameter values of Table 3.3, we get the following buffer requirement $B(D)$ for the three different striping techniques (Table 3.4):

Striping Algorithm	$B(D)$
CGS	$D \cdot Q_d \cdot b_{dr} = Q \cdot b_{dr}$
MGS	$C \cdot D_c \cdot Q_c \cdot b_{dr} = Q \cdot D_c \cdot b_{dr}$
FGS	$Q \cdot D \cdot b_{dr}$

Table 3.4: Buffer requirement for CGS, MGS, and FGS

From the buffer requirement formulas of Table 3.4, we observe that for a given disk retrieval block size:

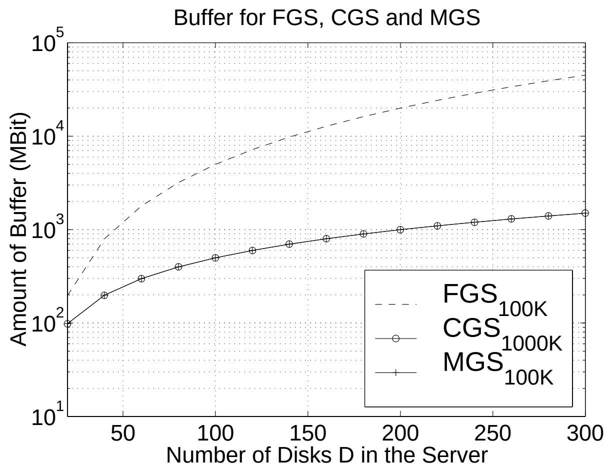
- For FGS: the total buffer is proportional to the product of the total number of disks D and the total number of clients Q.
- For CGS: the total buffer is only proportional to the total number of clients Q.
- For MGS: the total buffer is proportional to the product of the number of disks in a retrieval group D_c and the total number of clients Q.

Results

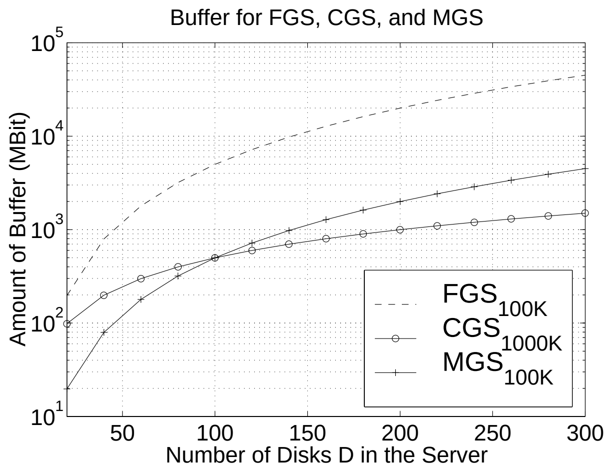
Let b_{dr}^{FGS}, b_{dr}^{CGS}, and b_{dr}^{MGS} b_{ru}^{FGS}, b_{ru}^{CGS}, b_{ru}^{MGS} denote respectively disk retrieval block and retrieval unit sizes of FGS, CGS, and MGS. We use the formulas of Table 3.4 to compute the buffer requirement for FGS, CGS and MGS.

Figure 3.9(a) shows the buffer requirement for FGS, CGS and MGS. We keep the retrieval group size D_c constant and vary the number of retrieval groups C for MGS. When the total number of disks increases, the number of retrieval groups increases while the retrieval unit size for MGS b_{ru}^{MGS} remains constant. However, the retrieval unit size for FGS b_{ru}^{FGS} grows with the increasing total number of disks. Since for CGS only one disk is involved to serve one client, the number of disks does not influence b_{ru}^{CGS}. We see that FGS requires much more buffer than CGS and MGS.

In Figure 3.9(b), we keep for MGS the number of retrieval groups constant ($C = 10$), and vary the number of disks D_c within one retrieval group. FGS results again in the highest buffer requirement. Since C is constant for MGS, the number of disks D_c per retrieval group grows when the total number of disks D increases. The buffer requirement for CGS and MGS follow respectively the formula: $B^{CGS} = Q \cdot b_{dr}^{CGS}$, and $B^{MGS} = Q \cdot b_{dr}^{MGS} \cdot D_c$ (Table 3.4). The increase of D_c for MGS results therefore in an increase in the amount of buffer required: For $D = 100$, we have $D_c = 10$, $B^{CGS} = Q \cdot 1000Kbit$, and $B^{MGS} = Q \cdot 100K \cdot 10 = Q \cdot 1000Kbit$. Therefore, for $D = 100$, we see in Figure 3.9(b) that $B^{MGS} = B^{CGS}$. For ($D < 100$), we have $D_c < 10$, and consequently $B^{MGS} < B^{CGS}$. For higher values of D ($D > 100$), we have $D_c > 10$, and consequently $B^{MGS} > B^{CGS}$.



(a) Buffer requirement for FGS, CGS, and MGS with $D_c = 10$ for MGS.



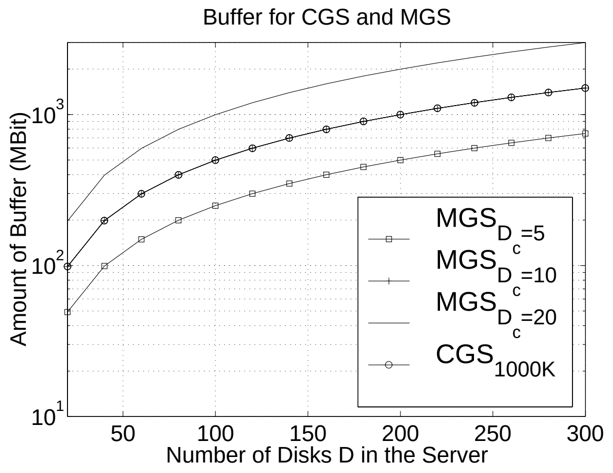
(b) Buffer requirement for FGS, CGS, and MGS with $C = 10$ for MGS.

Figure 3.9: Buffer requirement for FGS, CGS and MGS with $b_{dr}^{FGS} = b_{dr}^{MGS} = 100$ kbit, $b_{dr}^{CGS} = 1$ Mbit, $r_p = 1.5$ Mbit/sec, $r_d = 40$ Mbit/sec, $Q^{FGS} = Q^{CGS} = Q^{MGS}$ for each value of D.

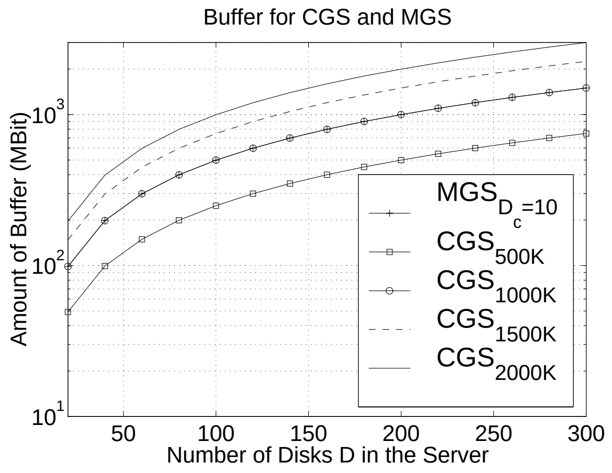
Figures 3.9(a) and 3.9(b) show that FGS requires the highest amount of buffer. Depending on the parameters b_{dr} and C, MGS or CGS has the lowest buffer requirement.

In the following we only compare the buffer requirement for CGS and MGS. Since the buffer requirement strongly depends on the choice of the retrieval unit sizes (b_{dr}^{CGS} and b_{dr}^{MGS}), we consider the following situations:

- We vary the retrieval group size ($D_c = 5, 10, 20$) and keep b_{dr}^{CGS} and b_{dr}^{MGS} constant (See Figure 3.10(a)). For MGS, $b_{ru}^{MGS} = D_c \cdot b_{dr}^{MGS}$ will vary with changing D_c.
- We vary the disk retrieval block size for CGS ($b_{dr}^{CGS} = 1, 1.5, 2$ Mbit) (See Figure 3.10(b)). Because $b_{dr}^{CGS} = b_{ru}^{CGS}$, we vary in this case the retrieval unit size for CGS.



(a) Buffer requirement for MGS and CGS with $D_c = 5, 10, 20$ for MGS and $b_{dr}^{MGS} = 100$ Kbit and $b_{dr}^{CGS} = 1$ Mbit.



(b) Buffer requirement for MGS and CGS for $b_{dr}^{CGS} = 0.5, 1, 1.5, 2$ Mbit and $b_{dr}^{MGS} = 100$ Kbit.

Figure 3.10: Buffer requirement for MGS and CGS for $b_{dr}^{MGS} = 100$ Kbit, $r_p = 1.5$ Mbit/sec, $r_d = 40$ Mbit/sec, $Q^{FGS} = Q^{CGS} = Q^{MGS}$.

Figures 3.10(a) and 3.10(b) show that the buffer requirement decreases when:

- For MGS: the size of the retrieval group D_c decreases, which implies a smaller retrieval unit size b_{ru}^{MGS},
- For CGS: the retrieval unit size b_{ru}^{CGS} decreases.

However, a decrease of the retrieval unit size b_{ru}^{CGS} for CGS will increase the seek overheads within disks, which results in a lower throughput, as we will see in section 3.3.2.

In order to reduce the buffer requirement for MGS, the retrieval group size D_c must decrease. However, a small retrieval group means a large number of retrieval groups for a given total number of disks D. We will see in section 3.3.3 that a small retrieval group size increases the latency for new client requests.

3.3.2 Server Throughput

To compare FGS, CGS, and MGS in terms of throughput, we use an admission control criterion calculating the maximum number of streams Q that can be admitted from a video server. The value of Q depends among others on the disk characteristics. In this Section, we will determine the throughput for each of the striping algorithms FGS, CGS, and MGS.

Admission Control Criterion

In order to avoid buffer starvation for all concurrent video streams, the time between the retrieval of two consecutive retrieval units should not exceed $\frac{b_{ru}}{r_p}$, which is the service round duration τ. Further, using SCAN as scheduling policy implies that disk heads travel across the disk surface twice in the worst case ($2 \cdot t_{seek}$). Additionally, the retrieval of data requires in the worst case the maximum value of the rotational latency (t_{rot}) (see section 2.2).

The admission control criterion computes the maximum number of streams that can be admitted. It takes into account the worst case latency overhead and the I/O bandwidth of the storage disks, the retrieval unit size and the video playback rate [OZDE 96b]. Since retrieval groups are independent from each other, we reduce the admission control discussion to a single retrieval group and derive later the formula for the video server depending on the striping algorithm used. According to the assumptions above and referring to section 2.2, the admission control criterion for a single disk is given by Eq. 3.1, where b_{ru} denotes the retrieval unit size of a stream. Now, we consider a video server with multiple disks and compute the its overall throughput depending on whether FGS, CGS, or MGS is used.

$$Q_d \cdot (\frac{b_{ru}}{r_d} + t_{rot}) + 2 \cdot t_{seek} = \frac{b_{ru}}{r_p} = \tau \tag{3.1}$$

For FGS, the retrieval unit of size b_{ru}^{FGS} is divided into exactly D disk retrieval blocks and each of that disk retrieval blocks is stored on a single disk. During one service round and for a single stream, each disk retrieves a disk retrieval block of size $b_{dr}^{FGS} = \frac{b_{ru}^{FGS}}{D}$. Since all server disks are involved to service each stream during every service round, FGS considers the whole video server as a single group. Hence the following admission control formula:

$$Q^{FGS} \cdot \left(\frac{\frac{b_{ru}^{FGS}}{D}}{r_d} + t_{rot} \right) + 2 \cdot t_{seek} = \frac{b_{ru}^{FGS}}{r_p}$$

Substituting b_{ru}^{FGS} by its value $D \cdot b_{dr}^{FGS}$, we then get:

$$Q^{FGS} \cdot \left(\frac{b_{dr}^{FGS}}{r_d} + t_{rot} \right) + 2 \cdot t_{seek} = \frac{D \cdot b_{dr}^{FGS}}{r_p} \tag{3.2}$$

For CGS, the entire retrieval unit of size b_{ru}^{CGS} is stored on a single disk and is therefore equivalent to the disk retrieval block as $b_{ru}^{CGS} = b_{dr}^{CGS}$. During one service round, one retrieval unit is retrieved from a single disk for each stream. Thus, CGS considers a single disk as a group that is independent of the other groups (disks). Therefore, the server throughput is simply calculated as the disk throughput Q_d^{CGS} times the total number of server disks D (Eq. C.2).

$$\begin{aligned} Q_d^{CGS} \cdot \left(\frac{b_{dr}^{CGS}}{r_d} + t_{rot} + t_{stl} \right) + 2 \cdot t_{seek} &= \frac{b_{dr}^{CGS}}{r_p} \\ \frac{Q^{CGS}}{D} \cdot \left(\frac{b_{dr}^{CGS}}{r_d} + t_{rot} \right) + 2 \cdot t_{seek} &= \frac{b_{dr}^{CGS}}{r_p} \end{aligned} \tag{3.3}$$

MGS divides the video server into C independent retrieval groups. During one service round, D_c disk retrieval blocks of size b_{dr}^{MGS} are retrieved for each stream from D_c different disks. The retrieval unit size for MGS is therefore $b_{ru}^{MGS} = D_c \cdot b_{dr}^{MGS}$. During two successive service rounds, a stream retrieves data from two consecutive retrieval groups. Hence, FGS is applied inside a single group, whereas CGS is used between the different retrieval groups of the server. Consequently, the overall server throughput Q^{MGS} equals the throughput Q_c^{MGS} of a single retrieval group times the number of retrieval groups C that are contained in the server and the admission control formula for MGS is depicted in Eq. C.4.

$$\frac{Q^{MGS}}{C} \cdot \left(\frac{b_{dr}^{MGS}}{r_d} + t_{rot} \right) + 2 \cdot t_{seek} = \frac{D_c \cdot b_{dr}^{MGS}}{r_p} \tag{3.4}$$

Note that Eq. C.4 covers the other two admission control formula of Eqs. C.3 and C.2. Indeed, If we take the values $C = 1$ and $D_c = D$, which correspond the the FGS algorithm, Eq. C.4 becomes equivalent to Eq. C.3. On the other hand, if we take the value $C = D$ and $D_c = 1$, which correspond to the CGS algorithm, Eq. C.4 becomes equivalent to Eq. C.2.

Let us now use Eqs. (C.3), (C.2), and (C.4) to derive the maximum number of admitted streams for FGS, CGS, and MGS illustrated in Table 3.5:

Results

We evaluate the throughput behavior for FGS, CGS and MGS. The disk parameters and their corresponding values are those of Seagate and HP for the SCSI II disk drives [GKSZ 96] and are depicted in Appendix A (Table A.1). We are based on the same scenario as in Figure 3.9(a).

Striping Algorithm	Maximum Number of Clients
FGS	$Q^{FGS} = \frac{\frac{D\cdot b_{dr}^{FGS}}{r_p} - 2\cdot t_{seek}}{\frac{b_{dr}^{FGS}}{r_d} + t_{rot}}$
CGS	$Q^{CGS} = \frac{\frac{b_{dr}^{CGS}}{r_p} - 2\cdot t_{seek}}{\frac{b_{dr}^{CGS}}{r_d} + t_{rot}} \cdot D$
MGS	$Q^{MGS} = \frac{\frac{D_c\cdot b_{dr}^{MGS}}{r_p} - 2\cdot t_{seek}}{\frac{b_{dr}^{MGS}}{r_d} + t_{rot}} \cdot C$

Table 3.5: Throughput for FGS, CGS and MGS.

Figure 3.11 shows how the throughput grows for FGS, CGS and MGS with an increasing number of disks in the server. CGS achieves the highest throughput, since CGS retrieves very large disk retrieval blocks during one service round ($b_{dr}^{CGS} = 1$ Mbit), and therefore keeps the total seek overhead low. For FGS and MGS, the disk retrieval blocks are much smaller ($b_{dr}^{FGS} = b_{dr}^{MGS} = 100$ Kbit) resulting in a higher seek overhead and a lower throughput.

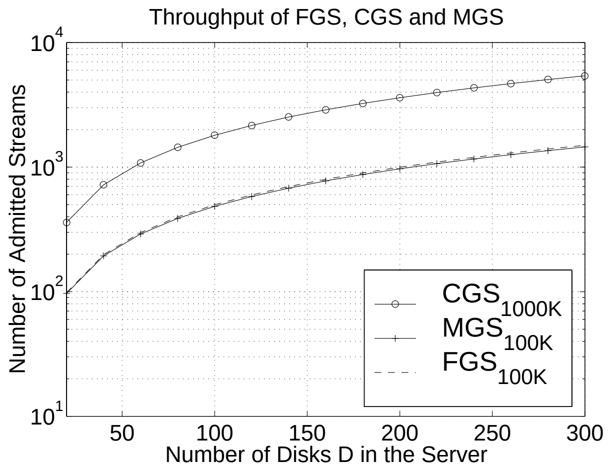
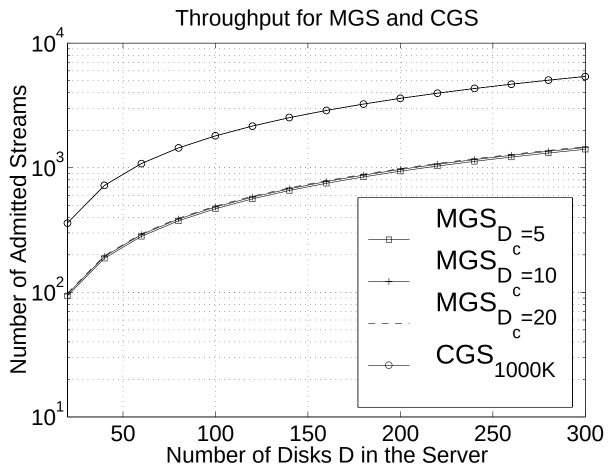


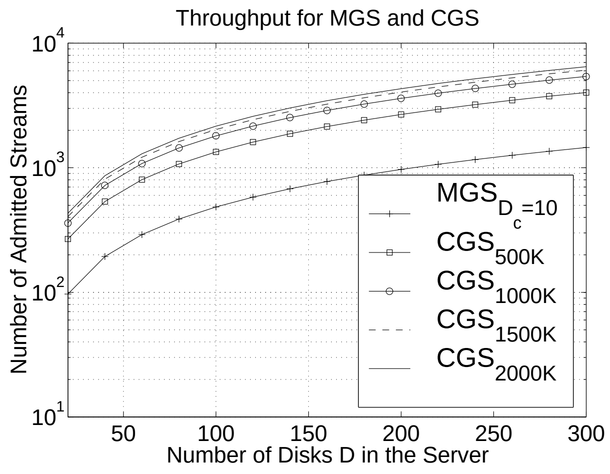
Figure 3.11: Throughput for FGS, CGS and MGS with $D_c = 10$ for MGS and $r_p = 1.5$ Mbit/sec.

Let us consider the results of Figures 3.9(a) and 3.3.11: Since $b_{dr}^{CGS} = 1$ Mbit and $b_{dr}^{FGS} = b_{dr}^{MGS} = 100$ Kbit for both Figures, we can compare FGS, CGS and MGS in terms of buffer requirement as well as throughput: We see that CGS has the best performance, since it has the same buffer requirement as MGS and the highest throughput. FGS requires much more buffer than CGS and MGS and admits fewer clients than CGS. For the same throughput, MGS requires less buffer than FGS.

Now we will only focus on the throughput comparison of CGS and MGS. Figures 3.10(a) and 3.10(b) have already shown that increasing retrieval units for CGS or for MGS increases the



(a) Throughput for CGS and MGS ($D_c = 5, 10, 20$) with $b_{dr}^{MGS} = 100$ Kbit and $b_{dr}^{CGS} = 1$ Mbit.



(b) Throughput for MGS and CGS with $b_{dr}^{CGS} = 0.5, 1, 1.5, 2$ Mbit and $b_{dr}^{MGS} = 100$ Kbit.

Figure 3.12: Throughput for MGS and CGS for $b_{dr}^{MGS} = 100$ Kbit, $r_p = 1.5$ Mbit/sec, $r_d = 40$ Mbit/sec.

amount of buffer. We will take the same parameter values and show the effect of varying the retrieval unit size for CGS and the retrieval group size for MGS on the throughput (Figures 3.12(a) and 3.12(b)).

We consider the results of Figures 3.10(a) and 3.12(a) in order to compare MGS and CGS in terms of both buffer requirement and throughput: MGS ($D_c = 10$ and $b_{dr}^{MGS} = 100$ Kbit) and CGS ($b_{dr}^{CGS} = 1$ Mbit) require the same amount of buffer. The throughput is however much higher for CGS than for MGS. The same results can be observed from Figures 3.10(b) and 3.12(b).

Figure 3.12(a) shows that the variation of the retrieval group size D_c for MGS has no significant influence on the throughput. On the other hand, Figure 3.12(b) shows that the throughput of CGS increases as the disk retrieval block size b_{dr}^{CGS} grows. However the buffer requirement increases too.

3.3.3 Start-up Latency

We consider the value $\tau = \frac{b_{ru}}{r_p}$ for the service round duration. An important performance measure, from the users point of view, is the **start-up latency** that is defined as the *maximum* elapsed time between the arrival of a new client and the retrieval of the first block for this client. Let τ^{FGS}, τ^{CGS}, and τ^{MGS} denote the duration of a service round for FGS, CGS, and MGS respectively. The corresponding start-up latencies are then T_s^{FGS}, T_s^{CGS} and T_s^{MGS}.

During one service round, a single disk retrieves multiple disk retrieval blocks for multiple streams. Each of these disk retrieval blocks is serviced during a slot of duration δ and τ is a multiple of δ. Let us call the slot that is not used for data retrieval a **free slot**. We compare FGS, CGS and MGS in terms of start-up latency for a given number of disks. We also look at the start-up latency behavior when the number of disks increases.

Start-up Latency for FGS

With FGS, all disks serve all video streams during each service round. When a new request arrives at disk d and there is a free slot, the retrieval of the corresponding video stream waits at most a service round τ^{FGS}:

$$T_s^{FGS} = \tau^{FGS} \tag{3.5}$$

We observe that T_s^{FGS} does not depend on the number of disks used and the requested disk. It also does not depend on the number of existing *free slots*.

Start-up Latency for CGS

We assume that a new request is arriving at disk d, with: $d \in [1..D]$ and the only existing *free slot* at this time is at disk $d+1$. The new request has to wait for $(D-(d+1)+d)$ service rounds, until the *free slot* attains disk d. Thus:

$$T_s^{CGS} = (D-1)\cdot\tau^{CGS} \tag{3.6}$$

T_s^{CGS} increases linearly when the total number of disks D increases.

Start-up Latency for MGS

We assume that a request is coming to group g with: $c \in [1..C]$ and the only existing *free slot* is at group $c+1$ at this time. To start retrieving data, the server has to wait for $(C-(c+1)+c)$ service rounds. Consequently, the worst case start-up latency is:

$$T_s^{MGS} = (C-1)\cdot\tau^{MGS} \tag{3.7}$$

Now, we want to compare T_S^{MGS} and T_S^{FGS}: If we assume that the sizes of a disk retrieval block are equal for FGS and CGS, then the retrieval unit size of FGS is a multiple of the the retrieval unit size of MGS, and consequently the service round τ^{FGS} is a multiple of τ^{MGS} as: $\tau^{MGS} = \frac{\tau^{FGS}}{C}$

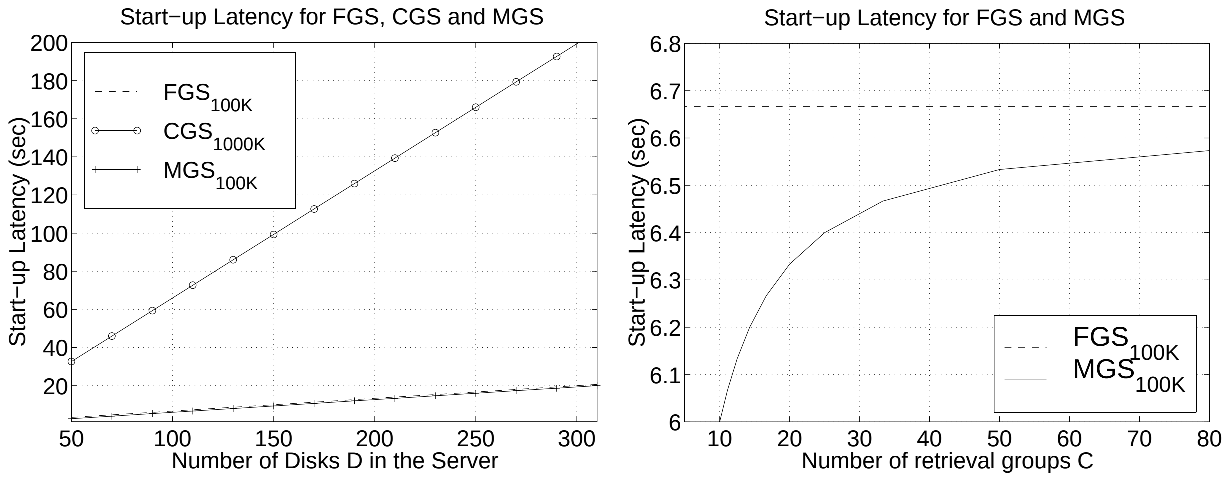
We derive the start-up latency of MGS: $T_s^{MGS} = (C-1)\cdot\tau^{MGS} = (C-1)\cdot\frac{\tau^{FGS}}{C} = \frac{(C-1)}{C}\cdot\tau^{FGS}$

T_s^{MGS} increases monotonously with larger values of C, but is always smaller than τ^{FGS}. T_S^{MGS} does not depend on the total number of disks in the server, but only on the number of retrieval groups.

Results

Figure 3.13(a) shows the variation of the start-up latency for FGS, CGS, and MGS. We vary the total number of disks between 10 and 200 and maintain a fixed number of disks inside a retrieval group (D_c = 10). The start-up latency is much higher for CGS than for MGS and FGS. The difference between T_s^{CGS} and T_s^{MGS} increases with an increasing number of disks. We also see that the start-up latency by MGS becomes increasingly closer to the one by FGS when the total number of disks D increases. This is due to an increase of C for a fixed D_c, when D increases.

Figure 3.13(b) considers only MGS and FGS. It shows how the worst case start-up latency of MGS depends on the number of retrieval groups C of the server for a given retrieval group size D_c and a fixed total number of disks D: When C grows, the start-up latency increases. For CGS, in order to decrease the start-up latency, we can reduce the service round duration. However, the throughput will then also decrease.



(a) Start-up latency for FGS, CGS and MGS ($D_c = 10$) with $b_{dr}^{FGS} = b_{dr}^{MGS} = 100$ kbit, $b_{dr}^{CGS} = 1$ Mbit

(b) Start-Up latency for MGS for different retrieval groups for $b_{dr}^{FGS} = b_{dr}^{MGS} = 100$ kbit and $D = 100$

Figure 3.13: Worst case start-up latency for $r_p = 1.5$ Mbit/sec.

3.4 Summary

In this chapter, we have classified data striping schemes for video servers. The classification is based on the striping and retrieval granularity of a video object (*video object striping*) and a video segment (*video segment striping*). Further, we have proposed the Mean Grained Striping (MGS) algorithm that divides the video server into independent groups, where all disks contained in the same node belong to different groups similarly to the orthogonal RAID configuration introduced in [GIBS 90]. The MGS algorithm covers the two other striping algorithms FGS and CGS as special cases if we choose the number of retrieval groups in the server to be 1 or D respectively. We have compared these three retained video wide striping algorithms FGS, CGS, and MGS in terms of buffer requirement, server throughput, and start-up latency. Our results demonstrate that for the same amount of buffer, CGS has a higher server throughput than MGS, while MGS in turn, achieves a higher throughput than FGS. With respect to the start-up latency, CGS is worst, whereas MGS has the lowest values. The results further indicate that FGS is the worst algorithm in terms of video server performance (buffer requirement for the same throughput or server throughput for the same amount of available buffer). Moreover, we have realized that the performance gap between CGS/MGS on one side and FGS on another side increases as the total number of server disks increases. In fact, as the number of disks in the server increases, the number of disk retrieval blocks to retrieve during one service round increases as well for FGS. This leads to an increase in the retrieval unit size, which in turn, results in a higher buffer requirement. If one were to keep the retrieval unit size the same for FGS regardless of the total number of disks, and thus keep the required buffer space low, then as the number of disks increases, the size of the disk retrieval block must be decreased correspondingly. As a result, the seek and rotational overhead becomes more important, which limits the number of concurrent streams (server throughput). Consequently, FGS is a *bad* data striping algorithm. Our further discussion will therefore be limited to CGS and MGS.

The results of this chapter indicate that CGS has highest server throughput for a *non fault-tolerant* video server. Making the video server fault-tolerant requires additional server resources in terms of disk storage, I/O bandwidth, and buffer capacity. Further, depending on the reliability scheme used, various striping algorithms may perform differently. The performance and quality of service of a fault-tolerant video server with respect to the striping algorithm and the reliability technique used is the focus of the next chapter.

Chapter 4

Data Striping and Server Reliability

The previous chapter investigated data striping for a non-fault tolerant video server. We found out that the Coarse Grained Striping algorithm (CGS) performs better than FGS and MGS in terms of the overall server throughput for a given amount of buffer. In this chapter, we introduce video server reliability and study the effect of making the video server fault-tolerant on server performance and quality of service depending on the striping algorithm used. We restrict our focus to the striping algorithms CGS and MGS and compare their server throughput, buffer requirement, and start-up latency for different reliability techniques.

The rest of this chapter is organized as follows. In section 4.1, we motivate server reliability for video servers and present the reliability techniques we consider; mirroring-based reliability and parity-based reliability. Section 4.2 presents related work. In section 4.3, we propose data layout for CGS and MGS for mirroring-based and for parity-based reliability and compare these different striping/reliability combinations in terms of server throughput, buffer requirement, and start-up latency. Section 4.4 finally summarizes our results.

4.1 Video Server Reliability

Even though the mean time to failure (MTTF) of a single disk is very high ,i.e. $300,000 \ \text{hours}$, a video server with, say, 200 disks has a MTTF of about $\frac{300{,}000}{200} = 1500 \ \text{hours}$ that is only $60 \ \text{days}$ [1]. Since the data contained on a failed disk is not accessible until the disk has been repaired, and given that each video object is distributed over *all* server disks (video object wide striping V_{ws}), a single disk failure affects *all* video objects stored and thus results in the interruption of service for all streams currently serviced. Hence, in order to provide continuous and uninterrupted service to all clients, it is imperative to reconstruct data that are residing on a

[1]We assume that life times of magnetic disk drives are exponentially distributed.

failed disk.

In the general context of disk arrays, schemes for ensuring the availability of data on failed disks have been proposed in the literature, e.g. [Patterson 88, CHEN 94a]. Most of these schemes are based on a RAID architecture and employ parity encoding to achieve fault-tolerance. However, these schemes assume conventional workload, in which reads access small amounts of data, are independent of each other, are aperiodic, and do not impose any real time requirements. In contrast, access to video data is sequential, periodic, and imposes real time constraints. Hence the need of reliability schemes that enable video data to be retrieved at a guaranteed rate despite disk failures as is the case for the streaming RAID scheme [TOB 93b] or the doubly striped scheme [MOUR 96].

For video server reliability, we will only consider schemes that add redundant data besides original data in order to make the video server fault-tolerant. Redundant data will be used to reconstruct original data that are contained on failed components[2]. There are two major techniques for a fault-tolerant video server that uses redundant data, a **mirroring-based** technique and a **parity-based** technique. In the following, we analyze for each of these two techniques how the placement of redundant data can be decisive with respect to the number of disk failures that are tolerated and the load balancing between server components during failure mode.

4.1.1 Mirroring-Based Reliability

Mirroring-based reliability consists in simply making a *copy* of each original video object [BITT 88, MERC 95, Mourad 96, BOLO 96, CHEN 97, HSDE 90, GOMZ 92]. Many researchers have proposed to replicate video data many times on the server, e.g. the replication scheme presented in [Korst 97], where the author proposes to store multiple copies of each original block and the choice of disks to store the replica is random. The result of this work shows that this approach has very small response times. In our work, however, we assume that a video object is *replicated once* on the video server and thus mirroring[3] results in 100% storage overhead. There are two ways of replicating a video object on a video server. The first way consists in replicating the whole original content of a given disk on a *separate* disk, called the *mirror* disk. We call this way of replication the **dedicated mirroring model**. The drawbacks of the dedicated mirroring model are (i) load-imbalances since the half of the server disks are idle during normal operation mode, whereas the other half may experience hot spot, and (ii) inefficient utilization of the server resources since up-to the half of the server bandwidth is not exploited. The second way of replicating a video object on a video server addresses these two problems by distributing replicated data besides original data on *all* server disks. We call this way of replication **shared mirroring model**. In [MOUR 96], the called doubly striped ap-

[2]The term component is used to denote a server disk or a server node.

[3]The term mirroring is used to denote mirroring-based reliability.

proach is proposed, where the secondary copy (replica) of each disk is uniformly distributed over all remaining $(D-1)$ disks of the video server. During the normal operation mode, each disk reserves a fraction of its available bandwidth to be used during failure mode. However, the doubly striped scheme does not tolerate more than a single disk failure, which may be insufficient for large video servers. To address this disadvantage and thus tolerate more than one disk failure, the authors in [BOLO 96] propose to distribute the secondary copy only over a *decluster* of d disks and not all remaining $(D-1)$ disks. However, each disk must reserve for failure mode a higher fraction of bandwidth than with the doubly striped approach. Note that the amount of bandwidth to be reserved and the number of disk failures that can be tolerated are conflicting. Indeed, as the value of d decreases to allow for more disk failures to be tolerated, the amount of bandwidth that must be reserved on each disk must be increased.

In the remainder of this thesis, We will limit our discussion to the shared mirroring model since it achieves higher throughput and better load balancing than the dedicated mirroring model.

4.1.2 Parity-Based Reliability

Parity-based reliability consists of storing *parity* data in addition to the existing *original* video data. In the general context of disk arrays, conventional RAID2–RAID6 schemes use parity-based reliability to protect the disk array against disk failures [LEE 92, CHAN 93, REDD 93, HOLL 94]. Parity-based reliability is also applied in the context of video servers similarly to the RAID architecture [TOB 93b, CLGK 94, GHAN 95a, COHE 96, OZDE 96a, TEWA 96b, BIRK 97]. Whereas mirroring-based reliability retrieves the copy of lost data that are residing on a failed disk, parity-based reliability uses parity data to reconstruct lost data. Parity data are encoded out of a set of original data, using typically a simple eXclusive-OR (XOR) function as shown in Figure 4.1 (A). The set of original data together with the parity data build the called **parity group**. No matter which original data of a parity group is lost, the surviving original data with the parity data are able to reconstruct the lost data by simply performing a XOR operation. The example in Figure 4.1 (B) illustrates this issue: The surviving original blocks 2, 3, and 4 together with the parity block P perform a XOR operation and reconstruct the lost original block 1.

Obviously, parity-based reliability requires a lower storage overhead, which is needed to store parity data, than mirroring-based reliability. As for replicated data, parity data can be placed on the server following two different ways. The first way consists in storing parity data on separate *parity* disks, which we call **dedicated parity model** [TOB 93b, BER 94a]. Thereby, parity disks are only used during failure mode, which creates load imbalances between server disks as well as inefficient use of server resources (bandwidth). The called **shared parity model** addresses this inefficiency by storing parity data on the same disks as original data, as studied in [COHE 96, GHAN 95a, OZDE 96a, CHEN 97, TEWA 96b, Kaddeche 98].

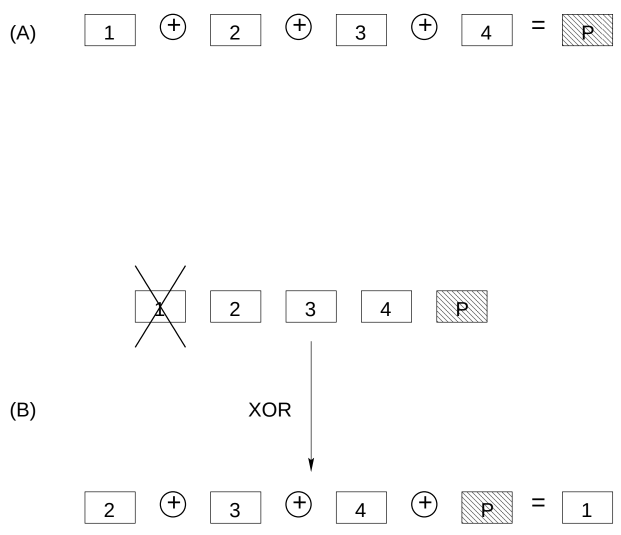


Figure 4.1: Parity encoding and decoding.

Let us consider the shared parity model and look at the layout alternatives of a parity group. If the parity group size is smaller than the total number of disks D in the server, then the placement of data belonging to different parity groups is decisive in terms of server performance, load balancing, and server reliability. The literature mainly distinguishes between (i) *independent*, also called *non-overlapping*, parity groups and (ii) *dependent*, also called *overlapping*, parity groups. In the case of independent parity groups, the video server is divided into separate parity groups, where each disk exclusively belongs to one parity group. Consequently, this alternative can tolerate one disk failure per parity group, which might be required to ensure high reliability for large video servers. However, since parity groups are independent of each other, a parity group that operates in failure mode may become a hot spot. Indeed, when a single disk fails within a parity group, the load of that failed disk is only distributed among the surviving disks of that parity group, which leads to load imbalances between the parity groups of the server. To address this inefficiency, schemes that use dependent (overlapping) groups are proposed. In [TEWA 96c], the called sequential parity placement is applied. It allows to distribute the load of a failed disk among two parity groups, which improves load-balancing when operating in failure mode. However, this improvement is traded for a reduced server reliability, since only one disk failure is tolerated per *two* parity groups. In [BIRK 97], the possibility for bottlenecks due to load-imbalances is alleviated by distributing blocks of a parity group *randomly* among the server disks. Therefore, the additional load generated by a failed disk is almost uniformly distributed among all surviving disks in the server. However, this random placement scheme only tolerates a single disk failure, which may be insufficient for large video servers. Further, with random placement algorithms, parity blocks may be placed within the same node as original blocks of the same parity group and hence node failures cannot be tolerated. Generally, the alternative of dependent parity groups (sequential and random placement) cannot provide the required reliability for large video servers. Consequently, we will consider in our further discussion only independent parity groups that can provide a high server reliability at the expense of some load-imbalances during failure mode.

To summarize, we consider for parity-based reliability the shared parity model. In the case where the size of the parity group is smaller than the total number of disks in the server, as the case for the MGS algorithm, we consider independent parity groups, where the video server is divided into equal size and non-overlapping parity groups.

4.2 Related Work

Fault-tolerance in a video server environment has received wide attention, see e.g. [COHE 96, GHAN 95a, OZDE 96a, CHEN 97, BIRK 97, BOLO 96, Mourad 96, TEWA 96b, Kaddeche 98]. In [BAAN 98], the authors compare RAID3 (FGS combined with parity) and RAID5 (CGS combined with parity) in terms of server performance and reliability. The results show that RAID5 performs better than RAID3. This work, however, only considers parity-based reliability and do not consider the MGS algorithm. In [TOB 93b], the Streaming RAID approach uses FGS/MGS with parity-based reliability. Thereby, the whole parity group is retrieved during a single service round, which allows to avoid any scheduling changes when switching from normal operation mode to failure mode. However, Streaming RAID has very high buffer requirement. The Staggered-group scheme in [BER 94a] addresses the high buffer requirement of Streaming RAID and reads a parity group during a single service round applying FGS/MGS, but plays it out during the next n service rounds, where n denotes the parity group size. The Software RAID approach [TEWA 96c] also uses parity-based reliability and combines FGS and CGS for data storage and retrieval. In fact, during normal operation mode, CGS is applied. During failure mode, FGS is only applied to reconstruct blocks that reside on the failed disk and CGS is applied otherwise. However, during failure mode, scheduling conflicts may arise during a switch from FGS retrieval to CGS retrieval.

In the context of mirroring-based reliability, The CGS algorithm is used in most of the literature: The doubly striped schemes proposed in [Mourad 96] replicates the original content of a single disk evenly among the other disks of the server. Thus, the load of a failed disk is balanced over all surviving server disks and each disk of them only needs to reserve a small fraction of its available bandwidth to be used during failure mode. However, the doubly striped approach tolerates only a single disk failure. Alternatively, the Microsoft Tiger video server [BOLO 96] replicates the original content of each disk across a subset of all disks and thus builds groups. Consequently, This approach tolerates more disk failures than the doubly striped scheme.

Most of the work cited above does not make explicit the relationship between data striping and reliability for a video server. Further, there is no direct comparison of parity-based reliability and mirroring-based reliability for a given data striping algorithm. This chapter investigates data striping and reliability aspects for video servers. We consider the two striping algorithms CGS and MGS and compare them in terms of server throughput, buffer requirement, and start-up latency in the case of mirroring-based reliability as well as parity-based reliability. Even though mirroring-based reliability doubles the storage volume required, it avoids the dramatic increase of the I/O bandwidth in case of failure that may be observed for parity-based reliability [HOLL 94]. In the following, we analyze this concern with respect to the data striping algorithm used.

4.3 Comparison of Striping and Reliability Schemes

The performance results of the last chapter show that CGS provides a higher throughput than MGS for a given amount of buffer. However, these results have only considered a non-fault tolerant video server. We now assume a fault-tolerant video server and compare MGS and CGS in terms of throughput for a given amount of buffer, worst case start-up latency for incoming client requests. We are interested in computing the start-up latency during failure mode. This start-up latency includes (i) the latency until a free slot is available as calculated in the last chapter and also (ii) the called **restart latency**. The restart latency for a given stream is defined as the worst case interval between the point of time where a single disk fails and the time at which the server is able to reconstruct the first disk retrieval block. As already mentioned, we focus on video server performance during both, normal operation mode and failure mode. Let us first determine the disk throughput during normal operation mode and during failure mode for CGS and MGS.

- *Normal operation mode*: During this mode, each disk should not exploit the entire available bandwidth. Instead, it keeps unused a part of the bandwidth, which reduces the maximum number of streams that can be admitted. Let us call Q_{no}^{CGS} and Q_{no}^{MGS} the maximum number of clients that can be serviced concurrently by a single disk that keeps unused a part of its bandwidth [4].

- *Disk Failure Mode*: When one out of the D (D_c) disks fails, the remaining $D-1$ (D_c-1) disks must support more streams than when operating in normal operation mode. The additional bandwidth load for each of the surviving disks is $\frac{Q_{no}^{CGS}}{D-1}$ for CGS and $\frac{Q_{no}^{MGS}}{D_c-1}$ for MGS. Thus, the maximum number of streams served in case of failure, Q_d^{CGS} and Q_d^{MGS} *per disk* is:

$$Q_d^{CGS} = Q_{no}^{CGS} + \frac{Q_{no}^{CGS}}{D-1} = Q_{no}^{CGS} \cdot \left(\frac{D}{D-1}\right) \tag{4.1}$$

[4]The index $_{no}$ denotes the normal operation mode.

$$Q_d^{MGS} = Q_{no}^{MGS} + \frac{Q_{no}^{MGS}}{D_c - 1} = Q_{no}^{MGS} \cdot (\frac{D_c}{D_c - 1}) \tag{4.2}$$

We use Eqs. 4.1 and 4.2 to calculate server throughput in the remainder of this chapter. These equations assume that in case of disk failure, the streams that would have been serviced from that disk are uniformly distributed over the $D - 1$ ($D_c - 1$) remaining disks. This assumption will be replaced in the next chapter by a worst case consideration, where the load of a failed disk may be entirely shifted to another disk.

4.3.1 Comparison for Mirroring-Based Reliability

We assume a shared mirroring model, where the secondary copy of each single disk is uniformly distributed over all remaining $(D - 1)$ disks for CGS, as proposed for the doubly striped scheme of Mourad [MOUR 96]. We propose in Figure 4.2 data placement of original disk retrieval blocks for the CGS algorithm. Figure 4.2 also illustrates, how original disk retrieval blocks that are stored on disk i are replicated across the other server disks.

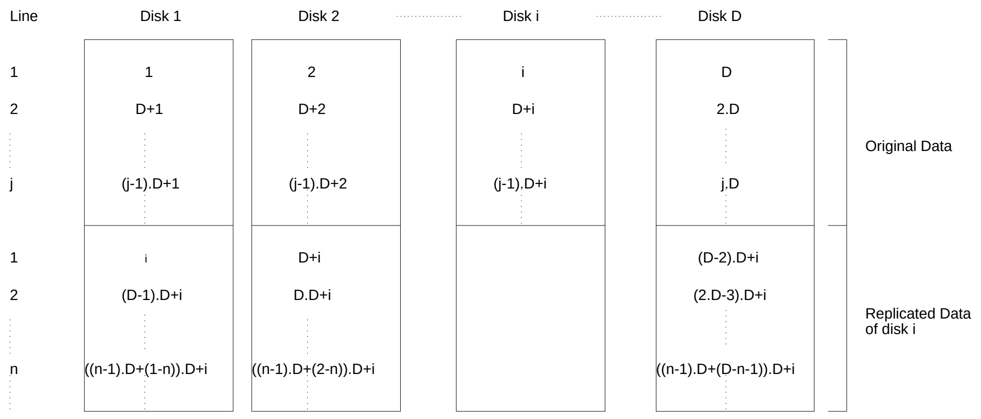


Figure 4.2: Mirroring for CGS.

For MGS, secondary data are stored among the remaining $(D_c - 1)$ disks of a single retrieval group. Figure 4.3 shows the placement of secondary copies among the remaining $(D_c - 1)$ disks of a retrieval group for MGS. Note that we only consider the first retrieval group (disks 1 to groups (2 to C). An extension to other retrieval groups is analogous.

Results

We compare CGS and MGS for mirroring-based reliability in terms of start-up latency and server throughput for the same amount of buffer.

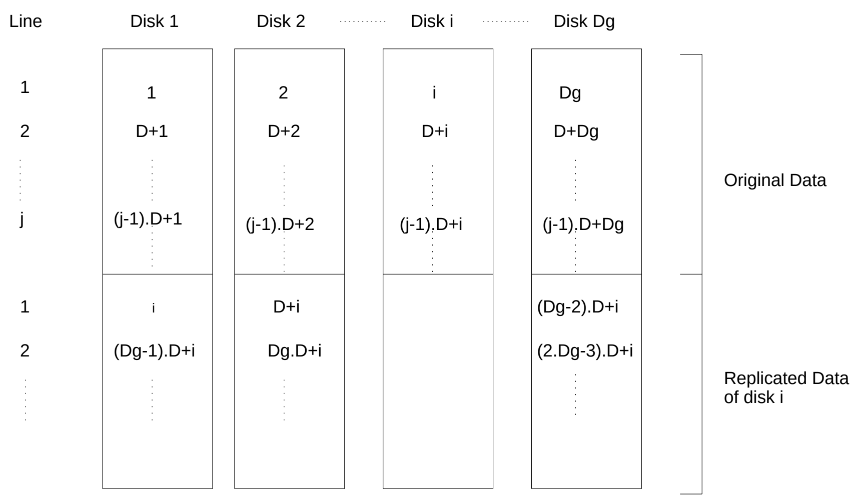


Figure 4.3: Mirroring for MGS for first retrieval group with disks 1 to D_c.

For a new incoming client during failure mode, the values of the worst case start-up latency for CGS and MGS are the same as during normal operation mode. In fact, during failure mode, the lost disk retrieval block is simply replaced by its replicated block, which does not require any additional delay. Consequently, for new clients and during failure mode, there is no restart latency with mirroring. We plot the start-up latency results in Figure 4.4(a). Note that these results are equivalent to those of Figure 3.13(a) in the last chapter.

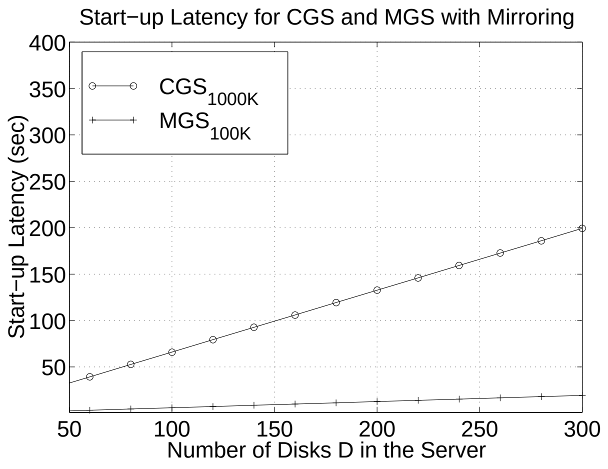
We refer to Eqs. (4.1) and (4.2) to compute the server throughput. When a disk fails, the original disk retrieval blocks residing on that failed disk are replaced when needed by their replica. These replica are retrieved from the surviving disks. As a result, the amount of buffer required for the video server does not depend on whether it is in normal operation or failure mode. Hence the throughput results in Figure 4.4(b) demonstrating that CGS still has a higher server throughput than MGS for the same amount of buffer when mirroring is used.

In summary, the results of Figure 4.4 indicate that for mirroring-based reliability, CGS, compared to MGS, keeps its throughput superiority for a given amount of buffer as for a non fault-tolerant video server. However, MGS has a lower start-up latency than CGS.

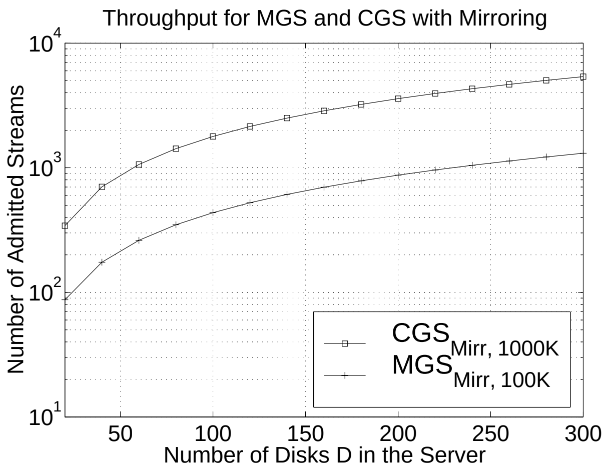
4.3.2 Comparison with Parity-Based Reliability

We now consider parity-based reliability, where parity disk retrieval blocks are used to reconstruct failed original disk retrieval blocks. We study the shared parity model, where parity data are distributed with original data on all D disks.

For CGS, $(D-1)$ *original* disk retrieval blocks and one *parity* disk retrieval block build one **parity group**. In Figure 4.5, we show how original and parity disk retrieval blocks of one video



(a) Start-up latency for CGS and MGS ($D_c = 10$).



(b) Throughput for CGS and MGS ($D_c = 10$) for the same amount of buffer.

Figure 4.4: Start-up latency and throughput for CGS- and MGS-based mirroring with $b_{dr}^{MGS} = 100$ kbit, $b_{dr}^{CGS} = 1$ Mbit and $r_p = 1.5$ Mbit/sec.

object are placed across the various disks of the video server. For that, we use a round robin data placement. The term ”P” denotes the parity disk retrieval block of a given parity group.

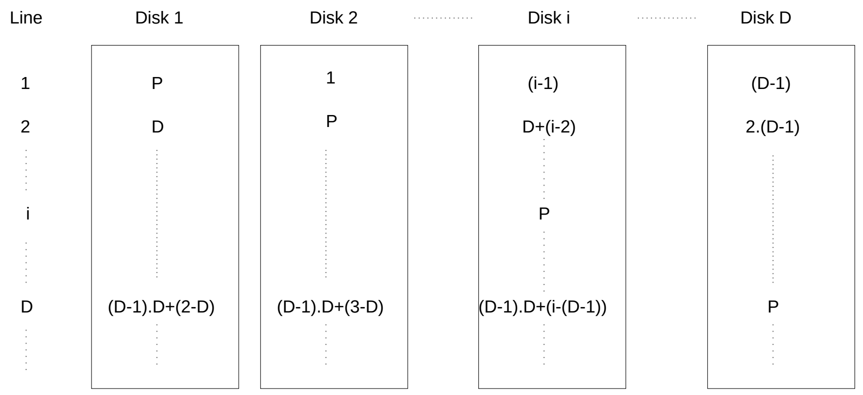


Figure 4.5: Parity data layout of the server for CGS.

Figure 4.6 shows within a single retrieval group the parity data placement for MGS. As for CGS, original disk retrieval blocks of one video object are stored in a round robin manner among all available D disks of the server. However, a parity group is built out of only $(D_c - 1)$ *original* disk retrieval blocks and one *parity* disk retrieval block. Like for mirroring, we only draw in Figure 4.6 the data layout of the first retrieval group (disks 1 to D_c) of the server.

Let us now compare MGS and CGS in terms of buffer requirements, server throughput, and

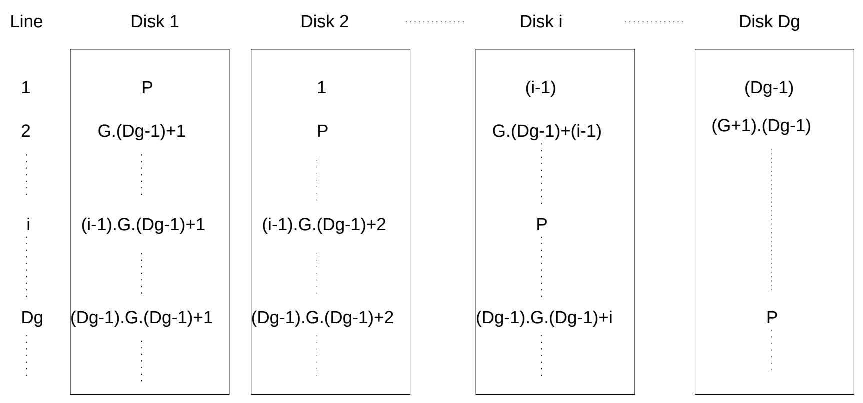


Figure 4.6: Parity data layout of the first retrieval group (disks 1 to D_c) for MGS.

start-up latency. In the context of parity decoding, we consider two ways of dealing with failure:

- **Reactive**: Means that parity information is only sent when a disk failure occurs, which implies that parity data are not used during normal operation mode.
- **Preventive**: Means that parity data are *always* sent with original information, even when working in normal operation mode. The bandwidth used for each of the surviving disks is therefore the same during both normal operation and failure mode.

Based on the distinction above, we discuss in the following four possible cases: MGS-reactive mode, MGS-preventive mode, CGS-reactive mode, and CGS-preventive mode.

MGS-reactive mode The $(D_c - 1)$ original disk retrieval blocks are sent for one stream during a service round. The parity disk retrieval block is only retrieved when a single disk fails inside the retrieval group. In this case, $(D_c - 2)$ original and one parity disk retrieval blocks are retrieved. Because we always (during normal operation and single disk failure mode) retrieve $(D_c - 1)$ disk retrieval blocks for the reactive mode, an admitted client requires the same amount of buffer and bandwidth from the video server before and after a single disk failure. However, the reactive mode can result in a transient degradation (for one service round).

MGS-preventive mode With this mode, parity information is automatically retrieved and sent with the original disk retrieval blocks. When working with normal operation mode, D_c disk retrieval blocks are sent. When a single disk fails inside a retrieval group, $(D_c - 1)$ blocks are sent, which are enough to reconstruct the missing video data. The advantage of the preventive mode is that there is neither a temporal degradation nor additional start-up latency when a single disk fails, as is the case with the reactive mode.

We should mention that parity data is per a retrieval group. Thus, each retrieval group can tolerate a single disk failure and the video server can tolerate multiple disk failures as well as a complete node failure (due to other component failures, i.e. operating system, hardware, cable).

CGS-reactive mode Only original disk retrieval blocks are retrieved using the CGS-reactive mode. Further, during normal operation mode, the buffer is immediately liberated after consumption. When a single disk fails, original as well as parity disk retrieval blocks are sequentially retrieved (during consecutive service rounds) from disks and temporarily stored in the buffer (for many service rounds) to reconstruct the lost original disk retrieval block. This requires additional buffer space. In the following, we calculate the amount of needed buffer and the worst case restart latency.

Assume a single disk failure is happening during service round $k-1$. At most, all Q_d^{CGS} disk retrieval blocks that should have been retrieved from this failed disk must be *reconstructed*. However, to reconstruct one failed disk retrieval block for one stream, $(D-1)$ disk retrieval blocks are *sequentially* retrieved (during $(D-1)$ successive service rounds) and *temporarily stored* in the buffer. We also call this strategy **buffering**. Q_d^{CGS} is as in Eq. (4.1).

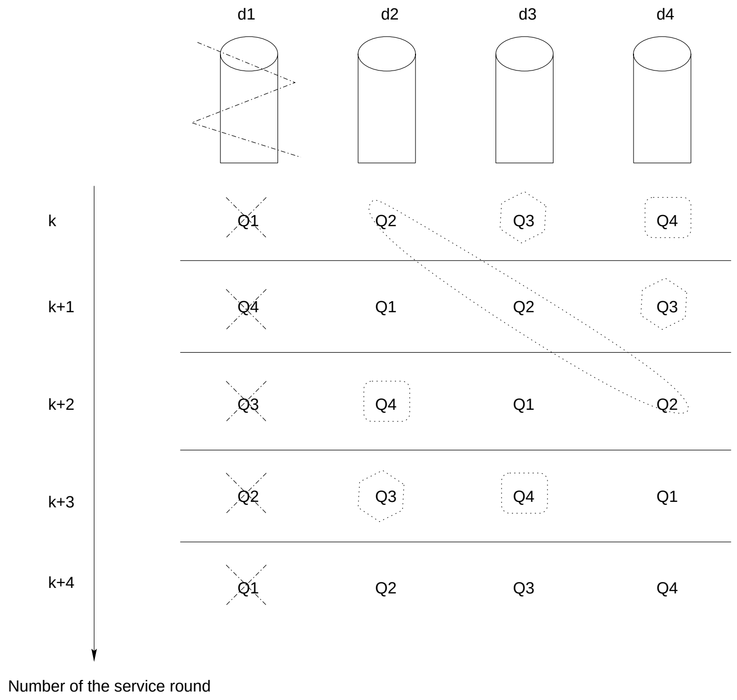


Figure 4.7: Latency for CGS after a disk failure.

The retrieval schedule of a CGS-parity-based server is depicted in Figure 4.7 for a simple scenario with 4 disks. $Q1$, $Q2$, $Q3$, and $Q4$ denote lists of clients. Each client is in exactly one list. Each list is served from one disk ($d1$, $d2$, $d3$, or $d4$) during one service round and from the next disk (round robin order) during the next service round. In Figure 4.7, we attribute to each of the lists ($Q1, Q2, Q3,$ and $Q4$) the corresponding disk ($d1$, $d2$, $d3$, or $d4$) from which data must be retrieved during service rounds k, $k+1$, $k+2$, $k+3$, and $k+4$. Let us assume that disk $d1$ fails during service round $k-1$ and let us focus on the data retrieval for clients in list $Q4$: During service round k, blocks are retrieved from disk $d4$; during service round $(k+1)$ no data is retrieved, since $d1$ has failed, during service round $(k+2)$ data is retrieved from disk $d2$, and during service round $(k+3)$ from disk $d3$. At the end of service round $(k+3)$, blocks of disk $d1$ can be reconstructed. Thus, streams belonging to list $Q4$ need four service rounds to reconstruct the failed information. For streams of lists $Q1$ and $Q3$, four service rounds are also needed, while streams of list $Q2$ take only three service rounds.

Figure 4.8 shows a single disk failure situation. A parity group contains 3 original disk retrieval blocks and one parity disk retrieval block (D_{rb}) . The first block is assumed to be lost. To reconstruct the whole stream, three times b_{dr}^{CGS} buffer space and four service round durations are required. Figure 4.8 illustrates the additional latency incurred and the amount of buffer space needed to reconstruct the missing data.

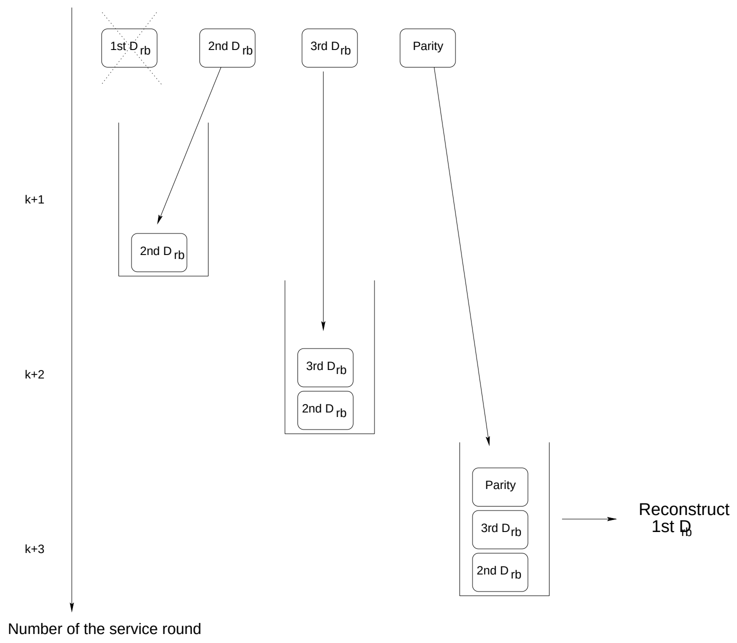


Figure 4.8: Buffer requirement for CGS during failure mode.

From Table 3.4, we know that the buffer requirement for a CGS based non-fault tolerant video server is: $B^{CGS}(D) = Q^{CGS} \cdot b_{dr}^{CGS}$, which is enough when working in *normal operation mode*.

For a fault-tolerant video server, the worst case buffer requirement needed for one stream is: $(D-1) \cdot b_{dr}^{CGS}$, since from the point of time where a disk failure occurs, the whole parity group of each stream must be kept in the buffer to reconstruct the lost block.

Thus, the buffer requirement for all Q^{CGS} streams during *single disk failure mode* is:

$$B^{CGS-buff}(D) = (D-1) \cdot Q^{CGS} \cdot b_{dr}^{CGS} = (D-1) \cdot B^{CGS}(D) \tag{4.3}$$

When a single disk fails, the restart latency varies between $(D-1) \cdot \tau^{CGS}$ and $D \cdot \tau^{CGS}$, depending on the placement of the disk retrieval blocks that belong to a parity group (Figure 4.7).

During failure mode, an admitted new client request needs to be delayed until free slots are available. Additionally, in the worst case, the client consumption must be delayed until the lost information is reconstructed. This additional delay is $(D-1) \cdot \tau^{CGS}$ and the total worst case start-up latency T_{rel}^{CGS} for CGS when working in disk failure mode is the sum of the worst case latency $T^{CGS} = (D-1) \cdot \tau^{CGS}$ and the additional delay:

$$T_{rel}^{CGS} = 2 \cdot (D-1) \cdot \tau^{CGS} \tag{4.4}$$

CGS-preventive mode To avoid temporal degradations for the admitted streams when a disk failure occurs, the video server can be preventive to be able to reconstruct the failed block at any time. This requires that blocks of a parity group be kept in the buffer even during normal operation mode. Thus, there is no difference between running in normal operation or disk failure mode in terms of buffer requirement. The overall needed amount of buffer is:

$$B^{CGS-buff}(D) = D \cdot Q^{CGS} \cdot b_{dr}^{CGS} = D \cdot B^{CGS}(D) \tag{4.5}$$

During normal operation mode, the parity information is not needed. In failure mode, parities will be needed to reconstruct a missing block.

The preventive mode eliminates or decreases the restart latency overhead produced by the reactive model, since some or all retrieval blocks of a parity group are already contained in the buffer when a missing retrieval block is needed and it takes less time to reconstruct the lost information. The throughput is as given in Eq. (4.1).

Eqs. (4.3) and (4.5) show that the buffer requirement *dramatically increases* (factor $(D-1)$ or D) for a CGS parity-based video server.

Results

In Figure 4.9, we plot the throughput and worst case start-up latency of CGS (buffering) and MGS when the video server is based on a parity and preventive model. Figure 4.9(a) shows that the start-up latency of CGS is becoming much higher than of MGS.

To compare the throughput for CGS (buffering) and MGS, we follow the next steps: Given the throughput for MGS for a non-fault-tolerant case (see Figure 3.11) [5], we derive the throughput for MGS with the parity-based scheme. Subsequently, we calculate the amount of buffer required to achieve this throughput. Finally, we calculate for CGS (buffering) the throughput that can be achieved given the same amount of buffer as for MGS.

We plot in Figure 4.9(b) the throughput for MGS and CGS in both cases (i) the non-fault-tolerant case (the two highest curves in the Figure) and (ii) the parity-based case (the two lowest curves in the Figure). The terms NF and Par in Figure 4.9(b) denote respectively the non-fault-tolerant case and the parity-based case. We observe that:

- The throughput for CGS decreases *enormously* when working with the parity-based scheme and the buffering strategy, compared with the non-fault-tolerant case. The decrease of the throughput is due to the buffer limitations that represent the bottleneck for CGS with the buffering strategy.

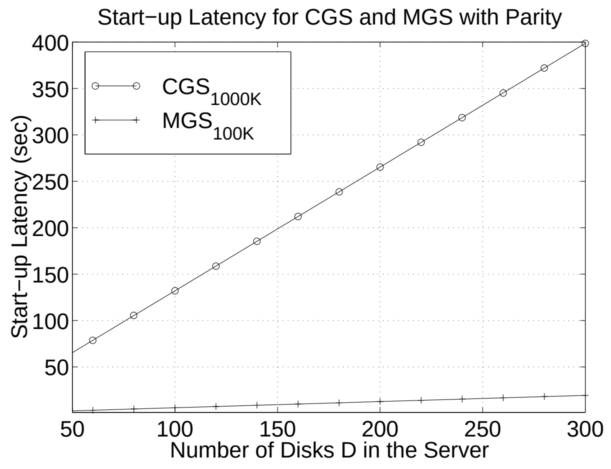
- The throughput of MGS with the parity-based scheme *slightly* decreases, compared with the non-fault-tolerant case.

- While CGS performs better than MGS in terms of throughput in the non-fault-tolerant case (the two highest curves of Figure 4.9(b)), MGS performs much better than CGS (buffering) in the parity-based case in terms of throughput (the two lowest curves of Figure 4.9(b)).

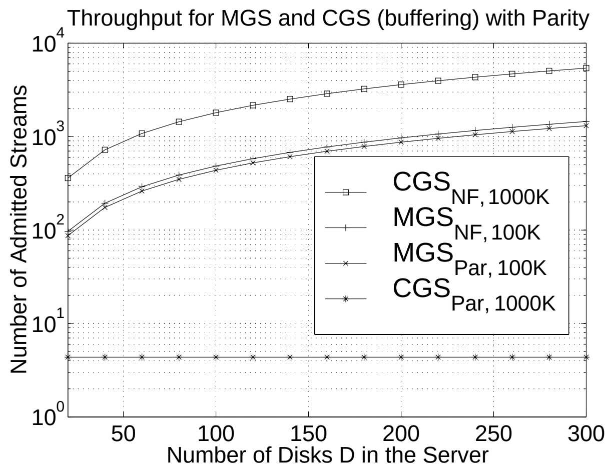
CGS-second read We saw that the buffer increases dramatically for a CGS-parity-based video server that uses the buffering strategy. Instead of temporarily storing all remaining disk retrieval blocks that belong to the same parity group, one can read every original disk retrieval block twice: one read to deliver the original block and another read to reconstruct the lost block. We call this method the **second read** strategy. Using a second read strategy, the number of reads will double and therefore the throughput will be cut in half ($Q_d^{CGS} = \frac{Q_{no}^{CGS}}{2}$). Further, an additional buffer is needed ($(D-1) \cdot Q_d^{CGS} \cdot b_{dr}^{CGS}$ for the reactive mode and $D \cdot Q_d^{CGS} \cdot b_{dr}^{CGS}$ for the preventive mode) to store data during the second read and perform decoding of the missing disk retrieval block. Let us only consider the preventive mode. Thus the total buffer requirement $B^{CGS-secread}(D)$ for the second read strategy is:

$$B^{CGS-secread}(D) = B^{CGS}(D) + D \cdot Q_d^{CGS} \cdot b_{dr}^{CGS} = 2 \cdot B^{CGS}(D) \qquad (4.6)$$

[5]We saw in Figure 3.11 that the throughput for CGS is higher than the one for MGS assuming the same amount of buffer.



(a) Start-up latency for CGS and MGS ($D_c = 10$).



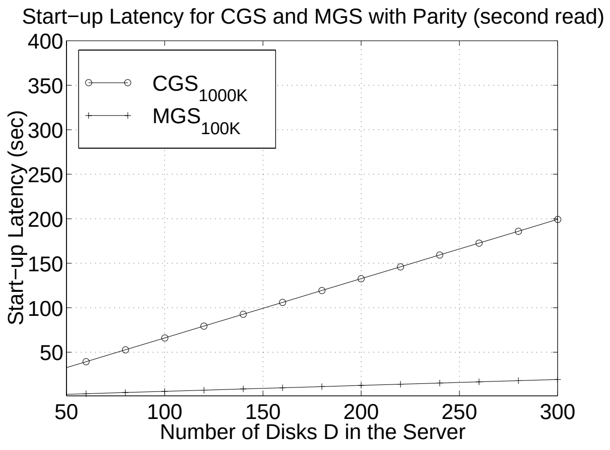
(b) Throughput for CGS and MGS for the same amount of buffer.

Figure 4.9: Start-up latency and throughput for CGS (*buffering*) and MGS with the preventive mode in a parity-based video server.

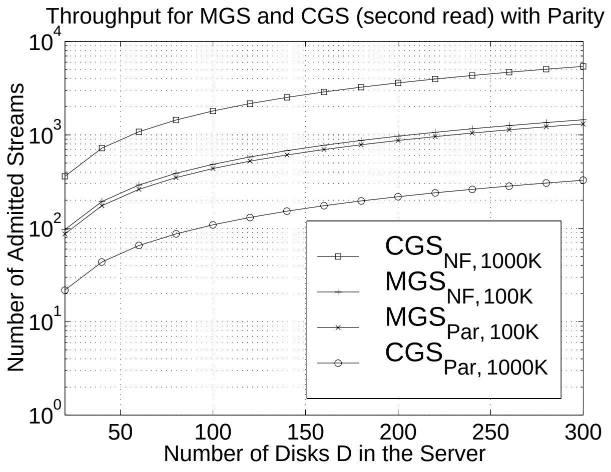
Unlike the buffering strategy, the second read strategy avoids the dramatic increase of the start-up latency and the restart latency, since disk retrieval blocks needed to reconstruct the missing block are simultaneously retrieved during one service round. Thus the worst case restart latency is one service round (τ^{CGS}) and the worst case start-up latency is the same as for CGS in a non-fault-tolerant server.

In Figure 4.10, we show the results of the worst case start-up latency and the throughput of CGS (second read) and MGS when the video server is based on a parity and preventive model. Figure 4.10(a) shows the decrease of the worst case start-up latency for CGS (second read) (compare with Figure 4.9(a)).

Analogous to Figure 4.9(b), we plot in Figure 4.10(b) the throughput for MGS and CGS in both cases, the non-fault-tolerant case (the two highest curves in the Figure), and the parity-based case (the two lowest curves in the Figure). The throughput results in Figure 10(b) show that the throughput has been improved for CGS with the second read strategy, compared with the throughput for CGS with the buffering strategy (Figures 4.9(b) and 4.10(b)). However, even with the second read strategy, CGS has a lower throughput than MGS using the parity-based technique. Compared with the results in Figure 4.9, we see that CGS with the second read strategy is better than CGS with the buffering strategy in terms of both, start-up latency and throughput. However, the implementation of the second read strategy is more complicated than the one of the buffering strategy. Indeed, the second read strategy alternates CGS retrieval for original not lost disk retrieval blocks and FGS retrieval when all remaining retrieval blocks of the parity group are read in parallel (during the same service round) to reconstruct the lost original disk retrieval blocks. The buffering strategy, however, always applies the CGS algorithm.



(a) Start-up latency for CGS and MGS ($D_c = 10$).



(b) Throughput for CGS (second read) and MGS for the same amount of buffer.

Figure 4.10: Start-up latency and throughput for CGS (*second read*) and MGS with the preventive mode in a parity-based video server.

4.3.3 Data Striping vs. Server Reliability: Discussion

We have compared for a fault-tolerant video server the two striping algorithms CGS and MGS with respect to the server throughput and the start-up latency. Our results indicate that for a mirroring-based video server, CGS has a higher server throughput than MGS for the same buffer space. For parity-based reliability, however, MGS has a higher throughput and a lower start-up latency than CGS. What remains is to decide whether (i) CGS & mirroring or (ii) MGS & parity is best for the video server. Before we make this decision, let us look at the potential bottleneck of a video server. We distinguish two types of servers: a *bandwidth-limited* and a *storage-limited* video server. Based on these two types, we argue in the following the choice of the best striping/reliability combination:

- For a bandwidth-limited video server, the bottleneck is the disk I/O bandwidth. Thus additional disks are only needed to provide the required I/O bandwidth, while their storage volume is not used. In this case, mirroring may not require any additional disks and a CGS mirrored video server will be the most cost-effective solution. The Tiger video server from Microsoft uses CGS and mirroring [BOLO 96].

- For the storage-limited case, the bottleneck is not the disk I/O bandwidth but the storage volume of the server. In this case, mirroring is not recommended because it will require doubling the storage volume, i.e. the number of disks. Instead, a MGS parity-based video server will be the most cost-effective solution.

4.4 Summary

Chapter 3 has compared data striping algorithms for non fault-tolerant video servers, where we have shown that the CGS algorithm has the highest overall server throughput for a given buffer space. In this chapter, we have introduced fault-tolerance within a video server. We have discussed mirroring-based and parity-based reliability and proposed for each of them its adequate data layout when both, CGS and MGS are considered. We have then compared CGS and MGS in terms of server throughput for the same amount of buffer and start-up latency when we use mirroring-based or parity-based reliability. Our results demonstrate that a video server based on mirroring has a higher server throughput with CGS than with MGS. For parity-based reliability, we have considered two modes of retrieving parity data: the reactive mode and the preventive mode. We have analyzed the video server behavior during disk failure for CGS and MGS. We found out that CGS (with the buffering strategy) suffers under very high buffer requirement when the server operates during disk failure and thus CGS has a much lower throughput than MGS given a fixed amount of buffer. The called second read strategy for CGS addresses this inefficiency by reading twice the needed original disk retrieval blocks instead of keeping them in the buffer, which is performed by the buffering strategy. As a result, the second read strategy reduces the amount of required buffer at the expense of cutting the server throughput to the half. The results show that MGS achieves a higher server throughput than CGS (with the buffering and the second-read strategy) for parity-based reliability. Finally, we have distinguished between bandwidth-limited and storage-limited video servers and argued the following conclusion: *For a bandwidth-limited video server, CGS combined with mirroring performs best, whereas for a storage-limited video server, MGS combined with parity performs best* [GABI 98c].

We consider in the remainder of the thesis a bandwidth-limited video server. This consideration relies on the fact that the main video server performance criterion is to serve as many clients as possible, which is the server throughput, and therefore the disk I/O bandwidth presents the most critical measure for the video server and is assumed to be the prime server bottleneck. Consequently, assuming a bandwidth-limited video server leads to make the choice for the CGS algorithm. For CGS, all of the terms retrieval unit, disk retrieval block, and video segment are equivalent and have the same size. We will simply use the term *block* to denote all of these terms. For the CGS algorithm, we present in Appendix B the storage layout of the original and redundant data and the data scheduling and retrieval procedures for operating during normal operation mode as well as during a single disk failure mode. We consider both, mirroring-based reliability and parity-based reliability.

Until now, we have for CGS only considered a parity group size that equals the total number of disks D in the server and the doubly striped mirroring scheme that distributes the original content of a single disk over *all* remaining server disks. Reducing (i) the parity group size or (ii) the number of disks, on which replicated data of a single disk are stored, may be decisive in terms of server reliability and performance. The cost, performance, and reliability of a fault-tolerant video server that is based on CGS are the focus of the next chapter.

Chapter 5

Video Server Reliability: Modeling and Performance

5.1 Introduction

We consider the striping algorithm CGS to store/retrieve original blocks on/from the video server. Making the video server fault-tolerant implies the storage of *redundant (replicated/parity)* blocks. What remains to be decided is *how* redundant data is going to be stored and retrieved. As indicated in the previous chapter, we only focus on shared mirroring/parity models that distribute redundant data (replica/parity) among all server disks in order to fairly distribute the load of the video server (see the previous chapter). Further, we consider the case where only original blocks of a video are used during normal operation mode. During failure mode, replicated/parity blocks are retrieved to reconstruct lost original blocks that reside on the failed component.

Let us first review the main characteristics of the shared mirroring and parity models. Mirroring (equivalent to RAID1 [LEED 93, CLGK 94]) consists in storing *copies* of the original data on the server disks and thus results in a 100% storage overhead. With shared mirroring (also called *interleaved declustering* [MERC 95]), original data and replicated data are spread over all disks of the server. On the other side, parity consists in storing parity blocks besides the original blocks. When a disk failure occurs, parity blocks together with the surviving original blocks of the same parity group are needed to reconstruct the content of the failed original block (see Figure 4.1). With shared parity, parity blocks are spread over all disks of the video server. The standard RAID5 [CHEN 94a] is a typical example of a shared parity scheme that is based on the CGS algorithm. Although the additional storage volume is small for parity-based reliability, the server needs additional resources in terms of I/O bandwidth or main memory when operating in failure mode. In the last chapter, we have distinguished for CGS combined with parity the *second-read strategy* and the *buffering strategy*. The second read strategy does not increase the buffer requirement when switching from normal operation mode to failure mode. However, this strategy sacrifices the half of the total available I/O bandwidth and therefore cuts the server throughput to the half. Instead, the buffering strategy reads each original block only once and thus optimizes the utilization of the I/O bandwidth (high throughput) at the expense of a high buffer requirement [GABI 98c]. We will restrict our discussion to the buffering strategy, since it achieves about twice as much throughput as the second read strategy [GABI 98c, GAFS 99a]. We will see that the buffering strategy becomes more attractive in terms of server performance (lower buffer requirement) and also regarding the server reliability as the size of the parity group decreases.

In this chapter we begin with a classification of several reliability schemes. in section 5.3, we present related work. Section 5.4 performs reliability modeling for the different schemes identified. Thereby, we split the discussion to the two cases of independent disk failures and dependent component failures. In this section, we also propose a novel mirroring scheme, called **Grouped One-to-One mirroring**, that achieves highest video server reliability. Video server performance (per stream cost) for the retained reliability schemes is discussed in section 5.5. Section 5.6 compares the different reliability schemes in terms of both, per stream cost and server reliability, and finally section 5.7 summarizes our results.

5.2 Classification of Reliability Schemes

Reliability schemes differ in the *technique* (parity/mirroring) and in the *distribution granularity* of redundant data. We define below the distribution granularity of redundant data.

- For the parity technique, the distribution granularity of redundant data is determined by whether the parity group comprises *All* (D) or *Some* (D_c) disks of the server. For the latter case, we assume that the server is partitioned into independent **groups** and that all groups are the same size, each of them containing D_c disks. Let C denote the number of groups in the server ($C = \frac{D}{D_c}$).
- For the mirroring technique, the distribution granularity of redundant data has two different aspects:
  - The first aspect concerns whether the original blocks of *one* disk are replicated on *One, Some* (D_c), or *All remaining* ($D - 1$) disks of the server.
  - The second aspect concerns how a *single original* block is replicated. Two ways are distinguished. The first way replicates *one* original block *entirely* into *one* replicated block [MOUR 96], which we call **entire block replication**. The second way

partitions *one* original block into *many sub-blocks* and stores each sub-block on a *different* disk [BOLO 96], which we call **sub-block replication**. We will show later on that the distinction between entire block and sub-block replication is decisive in terms of server performance (throughput and per stream cost).

Table 5.1 classifies mirroring and parity schemes based on their distribution granularity. We use the terms **One-to-One**, **One-to-All**, and **One-to-Some** to describe whether the distribution granularity of redundant data concerns *one* disk (mirroring), *all* disks (mirroring/parity), or *some* disks (mirroring/parity). For the One-to-One scheme, only mirroring is possible, since One-to-One for parity would mean that the size of each parity group equals 2, which consists in replicating each original block (mirroring). Hence the symbol ”XXX” in the table.

	Mirroring	**Parity**
One-to-One	Chained declustering [HSDE 90, GOMZ 92]	XXX
One-to-All	Entire block replication [MOUR 96, Mourad 96] Sub-block replication [MERC 95]	RAID5 with one group [CHEN 94a]
One-to-Some	Entire block replication Sub-block replication [BOLO 96]	RAID5 with many groups [BER 94a, TEWA 96a, OZDE 96a]

Table 5.1: Classification of the different reliability schemes

Table 5.1 distinguishes seven schemes. We will give for each of these schemes an example of the data layout . Thereby, we assume that the video server contains 6 disks and stores a single video. The stored video is assumed to be divided into exactly 30 *original* blocks. All schemes store original blocks in the same order (round robin) starting with disk 0. What remains to describe is the storage of redundant data for each of the schemes.

We present in the following examples of the mirroring-based schemes. These schemes have in common that each disk is partitioned into two separate parts, the first part storing original blocks and the second part storing replicated blocks.

As illustrated in Figure 5.1, the One-to-One mirroring scheme ($Mirr_{one}$) simply replicates original blocks of one disk onto another disk. If one disk fails, its load is *entirely* shifted to its *replicated* disk, which creates load-imbalances within the server (the main drawback of the One-to-One scheme).

In order to distribute the load of a failed disk evenly among the remaining disks of the server, the One-to-All mirroring scheme is applied as shown in Figures 5.2 and 5.3. Figure 5.2 depicts entire block replication ($Mirr_{all-entire}$) and Figure 5.3 depicts sub-block replication ($Mirr_{all-sub}$). In Figure 5.3, we only show how original blocks of disk 0 are replicated over

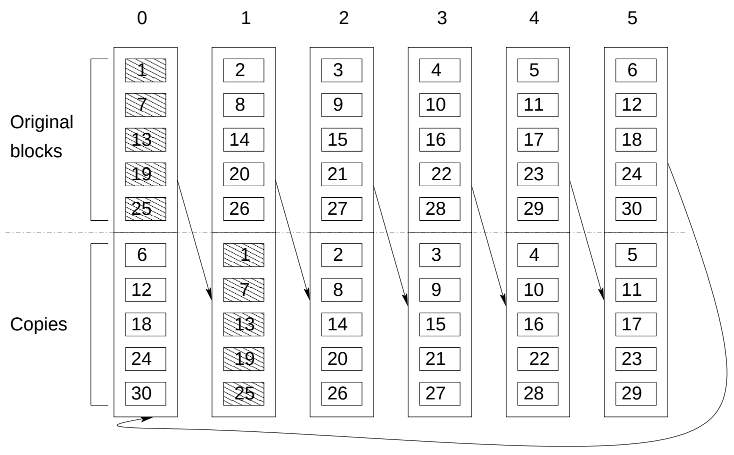


Figure 5.1: One-to-One Organization $Mirr_{One}$.

disks 1, 2, 3, 4, and 5. If we look at these two Figures, we realize that only a single disk failure is allowed. When two disk failures occur, the server cannot ensure the delivery of all video data.

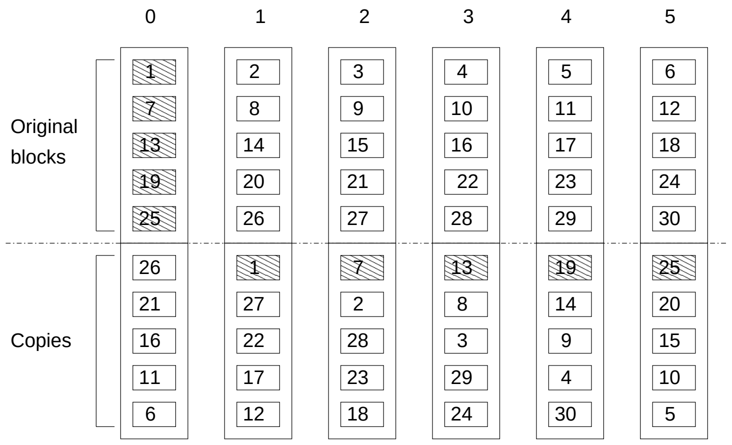


Figure 5.2: One-to-All Organization with Entire block replication $Mirr_{all-entire}$.

The One-to-Some mirroring scheme trades-off load-imbalances of the One-to-One mirroring scheme and the low reliability of the One-to-All mirroring scheme. Indeed, as shown in Figures 5.4 (entire block replication, $Mir_{some-entire}$) and 5.5 (sub-block replication, $Mir_{some-sub}$), the server is divided into multiple (2) independent groups. Each group locally employs the One-to-All mirroring scheme. Thus, original blocks of one disk are replicated on the remaining disks of the group and therefore the load of a failed disk is distributed over all remaining disks of the group. Further, since each group tolerates a single disk failure, the server may survive multiple disk failures.

We present next two layout examples of parity-based reliability. These correspond to the One-to-All parity scheme, Par_{all} (Figure 5.6) and the One-to-Some parity scheme, Par_{some} (Figure

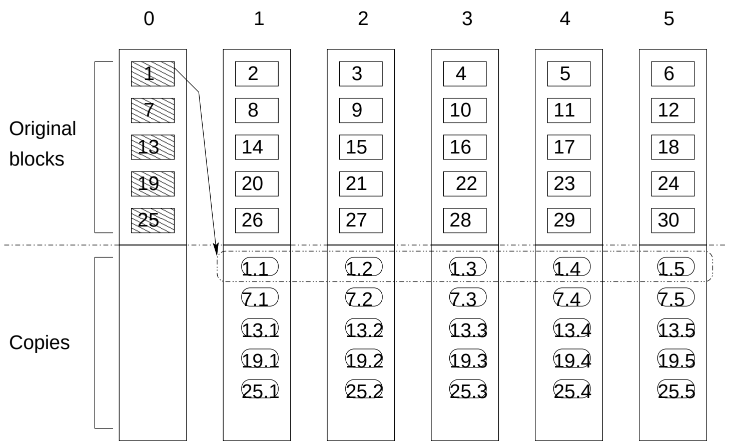


Figure 5.3: One-to-All Organization with Sub-blocks replication $Mirr_{all-sub}$.

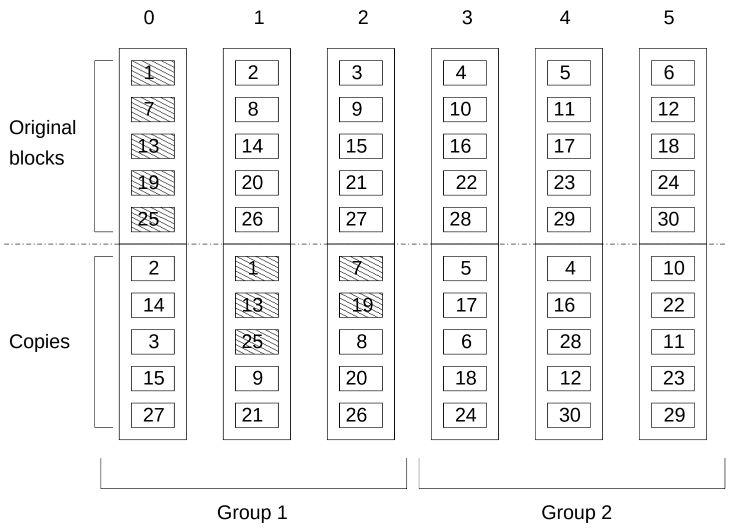


Figure 5.4: One-to-Some Organization with Entire block replication $Mirr_{some-entire}$.

5.7). In Figure 5.6 the parity group size is 6, e.g. the 5 original blocks 16, 17, 18, 19, and 20 and the parity block P_4 build a parity group. In Figure 5.7 the parity groups size is 3, e.g. the 2 original blocks 17 and 18 and the parity block P_8 build a parity group.

Looking at Figures 5.1 to 5.7, we observe that all One-to-All schemes (mirroring with entire block replication ($Mirr_{all-entire}$), mirroring with sub-block replication ($Mirr_{all-sub}$), and parity with one group (Par_{all})) only tolerate one disk failure. All these schemes therefore have the same server reliability. The same property holds for all One-to-Some schemes (mirroring with entire block replication ($Mirr_{some-entire}$), mirroring with sub-block replication ($Mirr_{some-sub}$), and parity with C groups (Par_{some})), since they all tolerate at most a single disk failure on each group. Consequently, it is enough for our reliability study to consider the three schemes (classes): One-to-One, One-to-All, and One-to-Some. For our performance

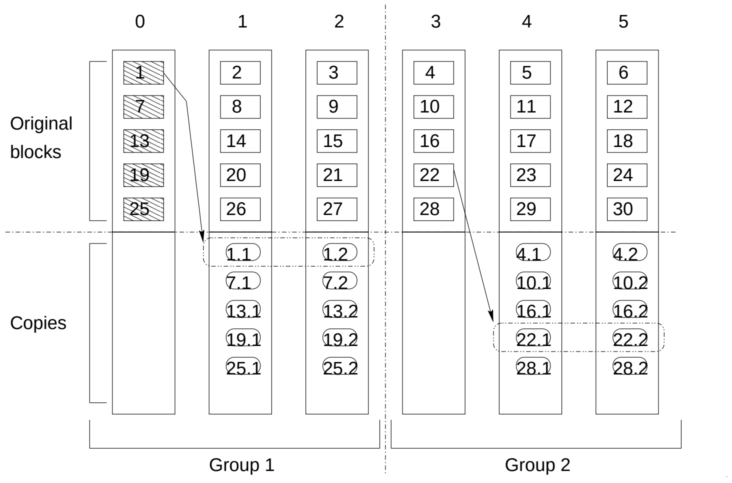


Figure 5.5: One-to-Some Organization with Sub-blocks replication $Mirr_{some-sub}$.

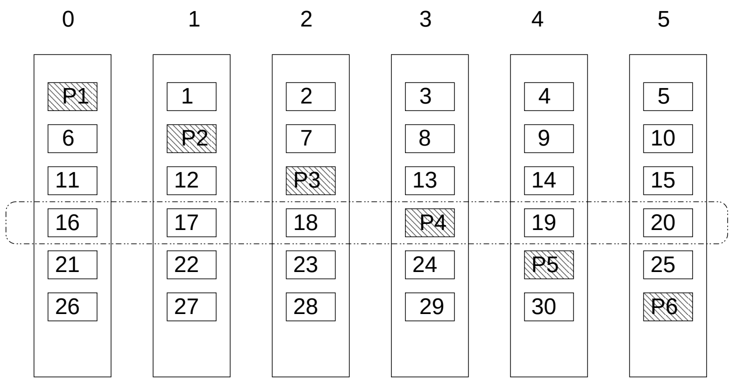


Figure 5.6: One-to-All Organization Par_{all}.

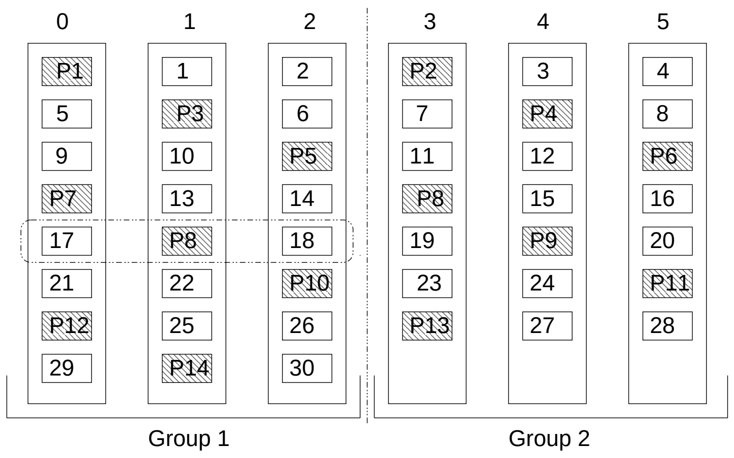


Figure 5.7: One-to-Some Organization Par_{some}.

study, however, we will consider the different schemes of Table 5.1 (see section 5.5).

5.3 Related Work

In the context of video servers, reliability has been addressed previously either by applying parity-based techniques similarly to the RAID2-RAID6 schemes, e.g. [HOLL 94, TEWA 96a, OZDE 96b, GOLP 98, GABI 98c], or by applying mirroring-based techniques similarly to the RAID1 scheme, e.g. [BITT 88, MOUR 96, BOLO 96]. However, none of these works has considered together all of the following aspects:

- Comparison of several parity-based and mirroring-based techniques under consideration of both, the video server performance and cost issues. Our cost analysis concerns the storage and the buffering costs to achieve a given server throughput.
- Reliability modeling based on the distribution granularity of redundant data in order to evaluate the server reliability for each scheme considered. We will perform a detailed reliability modeling that incorporates the case of independent disk failures and the case of dependent component failures.
- Performance, cost, and reliability trade-offs of different parity-based as well as mirroring-based techniques. We will study the effect of varying the group size on the server reliability and the per stream cost and determine the best value of the group size for each technique.

An extensive amount of work has been carried out in the context of parity-based reliability, see e.g. [LEE 92, TOB 93b, CLGK 94, HOLL 94, GHAN 95a, COHE 96, OZDE 96a, TEWA 96b, BIRK 97]. These contributions ensure a reliable real-time data delivery, even when one or some components fail. They differ in the way (i) they stripe data such as RAID3 (based on FGS) or RAID5 (based on CGS), and (ii) allocate parity information within the server (dedicated, shared, declustered, randomly, sequentially, etc.), and (iii) the optimization goals (throughput, cost, buffer requirement, load-balancing, start-up latency for new client requests, disk bandwidth utilization, etc.).

Video servers using mirroring have been proposed previously, see e.g. [BITT 88, MERC 95, Mourad 96, BOLO 96, CHEN 97, HSDE 90, GOMZ 92]. However, no *reliability modeling* has been carried out. Many mirroring schemes were compared by Merchant et al. [MERC 95], where some striping strategies for replicated disk arrays were analyzed. Depending on the striping granularity of the original and the replicated data, they distinguish between the uniform striping (CGS for original and replicated data in dedicated or in chained form) and the dual striping (original data are striped using CGS and replicated data are striped using FGS). However, their work is different from our study in many regards. First, the authors assume that both copies are used during normal and failure operation mode. Second, the comparison of different mirroring schemes is based on the *mean response times* and on *the throughput* achieved without taking into account server reliability. Finally, the authors do not analyze the impact of varying the distribution granularity of redundant data on server reliability and server performance.

In a general context of RAID, Trivedi et al. [MATR 93] analyzed the reliability of RAID1–RAID5 and focused on the relationship between disk's $MTTF$ and system reliability. Their study is based on the assumption that a RAID system is partitioned into *cold* and *hot* disks, where only hot disks are active during normal operation mode (dedicated model). Instead, we study reliability strategies for video servers that do not store redundant data separately on *dedicated* disks, but distribute original and redundant data evenly among *all* server disks (shared model). Gibson [GIBS 90] uses continuous Markov models in his dissertation to evaluate the performance and reliability of parity-based redundant disk arrays.

In the context of video servers, reliability modeling for *parity-based schemes* (RAID3, RAID5) has been performed in [BAAN 98] and RAID3 and RAID5 were compared using Markov reward models to calculate server availability. The results show that RAID5 is better than RAID3 in terms of the so-called performability (availability combined with performance).

To the best of our knowledge, there is no previous work in the context of video servers that (i) has compared several mirroring and parity schemes with various distribution granularities in terms of the server reliability and the server performance and costs and (ii) focused on the per stream cost and server reliability trade-offs with respect to the group size for parity as well as for mirroring.

5.4 Reliability Modeling

5.4.1 Motivation

The *server reliability* (also called the reliability function) $R^s(t)$ at time t is defined as the probability that the server survives until time t assuming that *all* server components are initially operational [STIEW 82]. The server survives as long as its working components deliver any video requested to the clients. As we have already mentioned, the server reliability depends on the distribution granularity of redundant data and is independent of whether mirroring or parity is used. In fact, what counts for the server reliability is the number of disks/nodes that are allowed to fail without causing the server to fail. As an example, the One-to-All mirroring scheme with entire block replication ($Mirr_{all-entire}$) only tolerates a single disk failure. This is also the case for $Mirr_{all-sub}$ and for Par_{all}. These three schemes have therefore the same server reliability. In light of this fact, we use the term One-to-All to denote all of them three for the purpose of our reliability study. Analogous to One-to-All, the term One-to-Some will represent the three schemes $Mirr_{some-entire}$, $Mirr_{some-sub}$, and Par_{some} and the term One-to-One denotes the One-to-One mirroring scheme $Mirr_{one}$.

We use Continuous Time Markov Chains (CTMC) for the server reliability modeling [SATR 96] . CTMC has discrete state spaces and continuous time and is also referred to as *Markov process* in [HOYL 94, Randolph 95]. Further, we exploit the strong empirical evidence that the failure rate λ_s of a system during its normal lifetime is approximately constant [STIEW 82, GIBS 90]. This is equivalent to an exponential distribution of the system's lifetime. In other terms, the system's reliability function $R^s(t)$ is:

$$R^s(t) = e^{-\lambda_s \cdot t}$$

Assuming that the system's mean time to failure $MTTF_s$ has the property: $MTTF_s < \infty$, $MTTF_s$ can be determined as [HOYL 94]:

$$MTTF_s = \int_0^\infty R^s(t)dt = \int_0^\infty e^{-\lambda_s \cdot t}dt = \frac{1}{\lambda_s} \tag{5.1}$$

Note that a system can be a single disk, a single node, or the whole video server. Accordingly, for a single disk, the mean time to disk failure $MTTF_d$ is to get as:

$$MTTF_d = \frac{1}{\lambda_d}$$

where λ_d presents the disk failure rate.

Analogously to failure rate, we assume that the repair rate μ_s of a system is exponentially distributed and thus the mean time to repair $MTTR_s$ can be expressed as: $MTTR_s = \frac{1}{\mu_s}$. For a single disk, the disk's mean time to repair is then

$$MTTR_d = \frac{1}{\mu_d}$$

where μ_d denotes the disk repair rate.

To build the *state-space diagram* [HOYL 94] of the corresponding CTMC, we introduce the following parameters: s denotes the total number of states that the server can have; i denotes a state in the Markov chain with $i \in [0..(s-1)]$; $p_i(t)$ is the probability that the server is in state i at time t. We assume that the server is fully operational at time $t_0 = 0$ and state 0 is the initial state. Additionally, state $(s-1)$ denotes the system failure state and is assumed to be an *absorbing state* (unlike previous work [BAAN 98], where a Markov model was used to compare the performance of RAID3 and RAID5 and allowed the repair of an overall server failure). When the server attains state $(s-1)$, it is assumed to stay there for an infinite time. Thus: $p_0(0) = 1,\ p_i(0) = 0\ \forall i \in [1..(s-1)]$, and $p_{(s-1)}(\infty) = 1$. The server reliability function $R^s(t)$ can then be computed as [HOYL 94]:

$$R^s(t) = \sum_{i=0}^{s-2} p_i(t) = 1 - p_{(s-1)}(t) \tag{5.2}$$

We present in the remainder of this section the Markov models for the three schemes One-to-All, One-to-One, and One-to-Some, assuming both, (i) independent disk failures (section 5.4.2) and (ii) dependent component failures (section 5.4.3).

5.4.2 Reliability Modeling for Independent Disk Failures

The One-to-All Scheme

With the One-to-All scheme ($Mirr_{all-entire}$, $Mirr_{all-sub}$, Par_{all}), data are lost if at least two disks have failed. The corresponding state-space diagram is shown in Figure 5.8, where states 0, 1, and F denote respectively the initial state, the one disk failure state, and the server failure state.

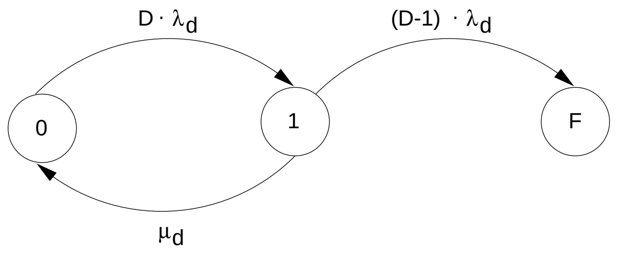


Figure 5.8: State-space diagram for the One-to-All Scheme.

The generator matrix Q_s of this CTMC is then:

$$Q_s = \begin{pmatrix} -D \cdot \lambda_d & D \cdot \lambda_d & 0 \\ \mu_d & -\mu_d - (D-1) \cdot \lambda_d & (D-1) \cdot \lambda_d \\ 0 & 0 & 0 \end{pmatrix}$$

The One-to-One Scheme

The One-to-One scheme ($Mirr_{one}$) is only relevant for mirroring. As the One-to-One scheme stores the original data of disk i on disk $((i+1)$ mod $D)$, the server fails if *two consecutive* disks fail. Depending on the location of the failed disks, the server therefore tolerates a number of disk failures that can take values between 1 and $\frac{D}{2}$. Thus, the number of disks that are allowed to fail without making the server fail can *not* be known in advance, which makes the modeling of the one-to-one scheme complicated: Let the D server disks be numbered from 0 to $D-1$. Assume that the server continues to operate after $(k-1)$ disk failures. Let $P^{(k)}$ be the probability that the server does not fail after the k^{th} disk failure. $P^{(k)}$ is also the probability that no disks that have failed are consecutive (adjacent). Before illustrating the state-space diagram for the One-to-One scheme, let us first calculate the probability $P^{(k)}$ that the video server does not fail after k disk failures for $k \in [1 \ldots \frac{D}{2}+1]$. It is obvious that the following holds: $P^{(1)} = 1$ and $P^{(\frac{D}{2}+1)} = 0$. Now, we consider the values $k \in [2..\frac{D}{2}]$. Theorem 5.4.1 gives the expected value of $P^{(k)}$ followed by the proof of this result. Thereby, the term C_k^n denotes the combinatorial function of the integers n an k as $C_k^n = \frac{n!}{k! \cdot (n-k)!}$.

Theorem 5.4.1

$$P^{(k)} = \frac{C_{k-1}^{n-1-k} + C_k^{n-k}}{C_k^n} \qquad \forall k \in [2 \cdots \frac{D}{2}]$$

Proof Let us consider the suite of n units. Note that the term unit can denote a disk (used for the independent disk failure case) or a node (used for the dependent component failure mode). Since we want to calculate the probability that the server does not fail after having k units down, we want those units not to be *adjacent*. Therefore we are looking for the sub-suites $i_1, i_2, \cdots, i_k \in [1, \cdots, n]$ such that:

$$\forall l \in [1..k-1], i_{l+1} - i_l > 1 \tag{5.3}$$

Let us call S the set of these suites. We introduce a bijection of this set of suites to the set $j_1, j_2, \cdots, j_k \in [1, \cdots, n-k+1]$ such that $j_1 = i_1, j_2 = i_2 - 1, j_3 = i_3 - 2, \cdots, j_k = i_k - (k-1)$.

Introducing the second suite j allows to suppress condition (5.3) on the suite i_l, since the suite j is strictly growing, whose number of elements are thus easy to count: it is the number of strictly growing functions from $[1..k]$ in $[1..n-(k-1)]$, that is $C_k^{n-(k-1)} = \frac{(n-k+1)!}{k! \cdot (n-2 \cdot k+1)!}$

This result though doesn't take into account the fact that the units number 1 and n are adjacent. In fact, two scenari are possible

- The first scenario is when unit 1 has already failed. In this case, units 2 and the n are not allowed to fail, otherwise the server will fail. We have then a set of $n-3$ units among which we are allowed to pick $k-1$ non-adjacent units. Referring to the case that we just solved, we obtain: $C_{k-1}^{n-3-(k-2)} = C_{k-1}^{n-k-1} = \frac{(n-k-1)!}{(k-1)! \cdot (n-2 \cdot k-2)!}$
- The second scenario is when the first unit still works. In this case, k units are chosen among the $n-1$ remaining ones. This leads us to the value $C_k^{n-1-(k-1)} = C_k^{n-k} = \frac{(n-k)!}{k! \cdot (n-2 \cdot k)!}$

The number of possibilities N_k that we are looking for is given by : $N_k = C_{k-1}^{n-1-k} + C_k^{n-k}$

Hence the results in theorem 5.4.1 that gives for a given number k of failed units (disks/nodes), the probability $P^{(k)}$ that the server does not fail after k unit failures.

$\square$

Figure 5.9 shows the state-space diagram for $Mirr_{one}$. If the server is in state i ($i \geq 1$) and failure $i+1$ happens, then the probability that the server fails (state F) equals $(1 - P^{(i+1)})$ and the probability that the server continuous operating (state $i+1$) equals $P^{(i+1)}$. The parameters of Figure 5.9 have the following values:

$\lambda_1 = D \cdot \lambda_d$, $\lambda_2 = (D-1) \cdot \lambda_d \cdot P^{(2)}$, $\lambda_3 = (D-2) \cdot \lambda_d \cdot P^{(3)}$, $\lambda_{n+1} = (D-n) \cdot \lambda_d \cdot P^{(n+1)}$, $A = (D-1) \cdot \lambda_d \cdot (1 - P^{(2)})$, $B = (D-2) \cdot \lambda_d \cdot (1 - P^{(3)})$, $G = (D-n) \cdot \lambda_d \cdot (1 - P^{(n+1)})$, $H = (D-(n+1)) \cdot \lambda_d \cdot (1 - P^{(n+2)})$, $Z = \frac{D}{2} \cdot \lambda_d$, and $\mu = \mu_d$.

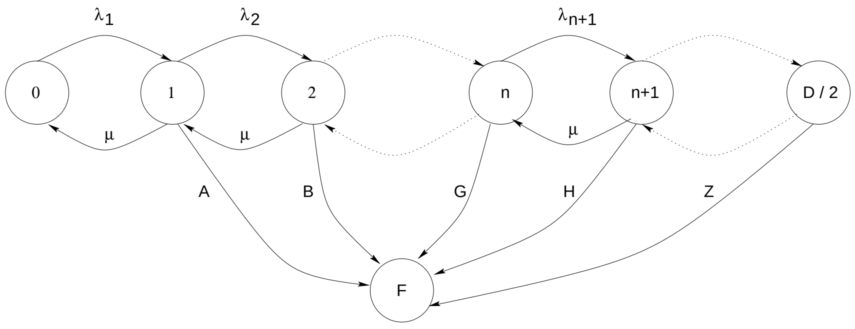


Figure 5.9: State-space diagram for the One-to-One scheme.

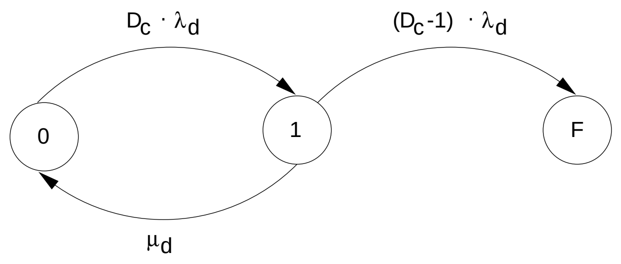
The corresponding generator matrix Q_s of the CTMC above is:

$$Q_s = \begin{pmatrix} -\lambda_1 & \lambda_1 & 0 & 0 & \cdots & 0 & 0 \\ \mu & -\mu - \lambda_2 - A & \lambda_2 & 0 & \cdots & 0 & A \\ 0 & \mu & -\mu - \lambda_3 - B & \lambda_3 & \cdots & 0 & B \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \mu & -\mu - Z & Z \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix} \quad (5.4)$$

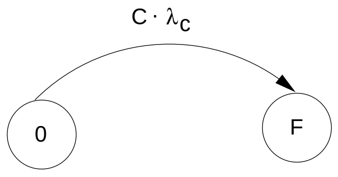
One-to-Some Scheme

The One-to-Some scheme (mirroring/parity) builds independent groups. The server fails if one of its C groups fails: We first model the group reliability and then derive the server reliability. Figures 5.10(a) and 5.10(b) show the state-space diagrams of one group and of the server respectively.

The generator matrix Q_c for the CTMC of a single group is:



(a) State-space diagram of one group.



(b) State-space diagram of the server.

Figure 5.10: State-space diagrams for the One-to-Some Scheme.

$$Q_c = \begin{pmatrix} -D_c \cdot \lambda_d & D_c \cdot \lambda_d & 0 \\ \mu_d & -\mu_d - (D_c - 1) \cdot \lambda_d & (D_c - 1) \cdot \lambda_d \\ 0 & 0 & 0 \end{pmatrix}$$

The group reliability function $R^c(t)$ at time t is $R^c(t) = \sum_{i=0}^{1} p_i(t) = p_{(0)}(t) + p_{(1)}(t) = 1 - p_{(2)}(t)$. The group mean lifetime $MTTF_c$ is then derived from $R^c(t)$. To calculate the overall server reliability function, we assume that the group failure distribution is exponential. The server failure rate is thus $C \cdot \lambda_c$, where C denotes the number of groups in the server and λ_c denotes the group failure rate ($\lambda_c = \frac{1}{MTTF_c}$). The server generator matrix Q_s of the CTMC of Figure 5.10(b) is therefore:

$$Q_s = \begin{pmatrix} -C \cdot \lambda_c & C \cdot \lambda_c \\ 0 & 0 \end{pmatrix} \tag{5.5}$$

5.4.3 Reliability Modeling for Dependent Component Failures

Dependent component failures mean that the failure of a single component of the server can affect other server components. We recall that our server consists of a set N of server nodes, where each node contains a set of D_n disks. Disks that belong to the same node have common components such as the node's CPU, the disk controller, the network interface, etc.. When any of these components fails, all disks contained in the affected node are unable to deliver video data and are therefore considered as failed. Consequently, a single disk does not deliver video data anymore if itself fails or if one of the components of the node fails to which this disk belongs. We present below the models for the different schemes for the case of dependent component failures. Similarly to a disk failure, a node failure is assumed to be repairable. The failure rate λ_n of the node is exponentially distributed with $\lambda_n = \frac{1}{MTTF_n}$, where $MTTF_n$ is the mean life time of the node. The repair rate μ_n of a failed node is exponentially distributed as $\mu_n = \frac{1}{MTTR_n}$, where $MTTR_n$ is the mean repair time of the node.

For mirroring and parity schemes, we apply the so called **Orthogonal RAID** mechanism whenever groups must be built. Orthogonal RAID was discussed in [GIBS 90] and [CLGK 94]. It is based on the following idea. Disks that belong to the same group must belong to different nodes. Thus, the disks of a single group do not share any (common) node hardware components. Orthogonal RAID has the property that the video server survives a complete node failure: When one node fails, all its disks are considered as failed. Since these disks belong to different groups, each group will experience at most one disk failure. Knowing that one group tolerates a single disk failure, all groups will survive and therefore the server will continue operating. Until now, Orthogonal RAID was only applied in the context of *parity* groups. We generalize the usage of Orthogonal RAID for both, mirroring and parity.

In order to distinguish between disk and node failure when building the models of the schemes considered, we will present each state as a tuple $[i, j]$, where i gives the number of disks failed and j gives the number of nodes failed. The failure (absorbing) state is represented with the letter F as before.

The One-to-All Scheme

For the One-to-All schemes ($Mirr_{all-entire}$, $Mirr_{all-sub}$, and Par_{all}), double disk failures are not allowed and therefore a complete node failure causes the server to fail. Figure 5.11 shows the state-space diagram for the One-to-All scheme for the case of dependent component failures. The states of the model denote respectively the initial state ($[0, 0]$), the one disk failure state ($[1, 0]$), and the server failure state (F).

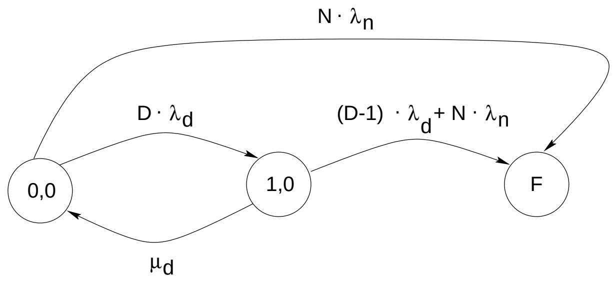


Figure 5.11: State-space diagram for the One-to-All scheme with dependent component failures.

The generator matrix Q_s is then:

$$Q_s = \begin{pmatrix} -D \cdot \lambda_d - N \cdot \lambda_n & D \cdot \lambda_d & N \cdot \lambda_n \\ \mu_d & -\mu_d - (D-1) \cdot \lambda_d - N \cdot \lambda_n & (D-1) \cdot \lambda_d + N \cdot \lambda_n \\ 0 & 0 & 0 \end{pmatrix}$$

The Grouped One-to-One Scheme

Considering dependent component failures, the One-to-One scheme as presented in Figure 5.1 would achieve a very low server reliability since the server immediately fails if a single node fails. We propose in the following an organization of the One-to-One scheme that tolerates a complete node failure and even $\frac{N}{2}$ node failures in the best case. We call the new organization the **Grouped One-to-One** scheme. The Grouped One-to-One organization keeps the initial property of the One-to-One scheme, which consists in *completely* replicating the original content of one disk onto another disk similarly to the chained declustering scheme. Further, the Grouped One-to-One organization divides the server into independent groups, where disks belonging to the same group have their replica inside this group. The groups are built based on the Orthogonal RAID principle and thus disks of the same group belong to different nodes as Figure 5.12 shows. Figure 5.12 assumes one video containing 40 original blocks and is stored on a server that is made of four nodes ($N_1, \cdots, N_4$), each containing two disks. Inside one group, up-to $\frac{D_c}{2}$ disk failures can be tolerated, where $D_c = 4$ is the number of disks inside each group. The Grouped One-to-One scheme can therefore survive $\frac{N}{2} = 2$ node failures in the best case (the server in Figure 5.12 continues operating even after nodes N_1 and N_2 fail). In order to distribute the load of a failed node among possibly *many* and not only *one* of the surviving nodes, the Grouped One-to-One scheme ensures that disks belonging to the same node have their replica on disks that do not belong to the *same* node [1].

In order to model the reliability of our Grouped One-to-One scheme, we first study the behavior of a single group and then derive the overall server reliability. One group fails when two consecutive disks inside the group fail. We remind that two disks are consecutive if the original data of one of these disks are replicated on the other disk (for group 1 the disks 0 and 4 are consecutive, whereas the disks 0 and 2 are not). Note that the failure of one disk inside the group can be due to (i) the failure of the disk itself or (ii) the failure of the whole node, to which the disk belongs. After the first disk failure, the group continues operating. If the second disk failure occurs inside the group, the group may fail or not depending on whether the two failed disks are consecutive. Let $P^{(2)}$ denotes the probability that the two failed disks of the group are not consecutive. Generally, $P^{(k)}$ denotes the probability that the group does not fail after the k^{th} disk failure inside the group. $P^{(k)}$ is calculated accordingly to theorem 5.4.1 in section 5.4.2.

[1] Assume that node N_1 has failed, then its load is shifted to node N_3 (replica of disk 0 are stored on disk 4) and to node N_4 (replica of disk 1 are stored on disk 7)

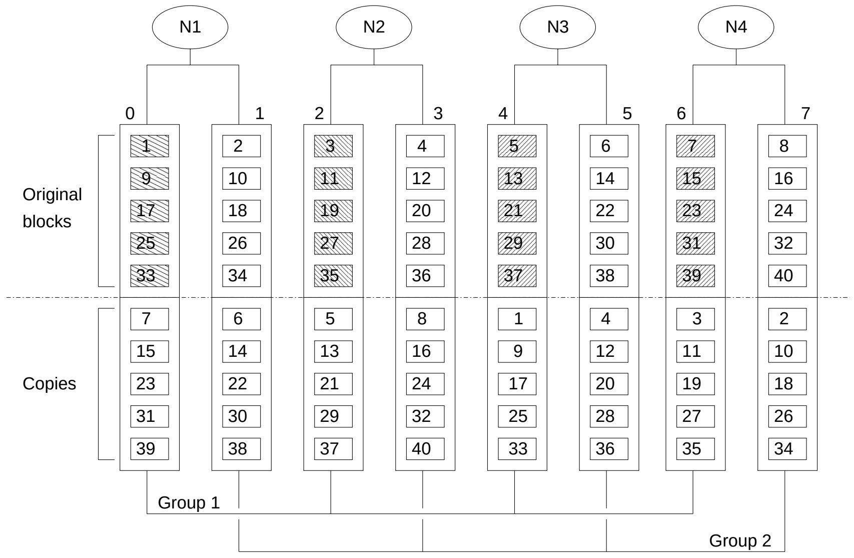


Figure 5.12: Grouped One-to-One scheme for a server with 4 nodes, each with 2 disks ($D_c = N = 4$).

The state-space diagram of one group for the example in Figure 5.12 is presented in Figure 5.13. The number of disks inside the group is $D_c = 4$. State $[i, j]$ denotes that $i + j$ disks have failed inside the group, where i disks, themselves, have failed and j nodes have failed. Obviously, all of the i disks that have failed belong to different nodes than all of the j nodes that have failed. We describe in the following how we have built the state-space diagram of Figure 5.13. At time t_0, the group is in state $[0, 0]$. The first disk failure within the group can be due to a single disk failure (state $[1, 0]$) or due to a whole node failure (state $[0, 1]$). Assume that the group is in state $[1, 0]$ and one more disk of the group fails. Four transitions are possible: (i) the group goes to state $[2, 0]$ after the second disk of the group has failed itself and the two failed disks are not consecutive, (ii) the group goes to state $[1, 1]$ after the node has failed, on which the second failed disk of the group is contained and the two failed disks are not consecutive, (iii) the group goes to state F after the second disk of the group has failed (disk failure or node failure) and the two failed disks are consecutive, and finally (iv) the group goes to state $[0, 1]$ after the node has failed, to which the first failed disk of the group belongs and thus the number of failed disks in the group does not increase (only one disk failed). The remaining of the state-space diagram is to derive in an analogous way. The parameters of Figure 5.13 are the following:

$\lambda_1 = D_c \cdot \lambda_d$, $\lambda_2 = N \cdot \lambda_n$, $\lambda_3 = (D_c - 1) \cdot \lambda_d \cdot P^{(2)}$, $\lambda_4 = (N - 1) \cdot \lambda_n \cdot P^{(2)}$, $F_1 = ((D_c - 1) \cdot \lambda_d + (N - 1) \cdot \lambda_n) \cdot (1 - P^{(2)})$, $\lambda_5 = (D_c - 2) \cdot \lambda_d \cdot P^{(3)}$, $\lambda_6 = (N - 2) \cdot \lambda_n \cdot P^{(3)}$, $F_2 = ((D_c - 1) \cdot \lambda_d + (N - 1) \cdot \lambda_n) \cdot (1 - P^{(2)})$, $\lambda_7 = (\frac{D_c}{2} \cdot \lambda_d) + (\frac{N}{2} \cdot \lambda_N)$, $\mu_1 = \mu_d$, and $\mu_2 = \mu_n$.

The generator matrix Q_c for a group is:

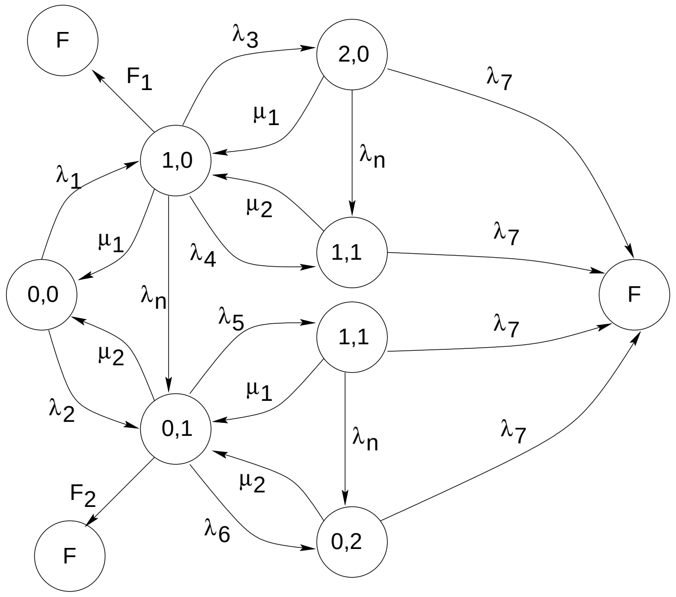


Figure 5.13: State-space diagram of one group for the Grouped One-to-One scheme with dependent component failures ($D_c = N = 4$).

$$Q_c = \begin{pmatrix} -\lambda_1 - \lambda_2 & \lambda_1 & \lambda_2 & 0 & 0 & 0 & 0 & 0 \\ \mu_1 & -A & \lambda_n & \lambda_3 & \lambda_4 & 0 & 0 & F_1 \\ \mu_2 & 0 & -B & 0 & \lambda_5 & \lambda_6 & F_2 & \\ 0 & \mu_1 & 0 & -\mu_1 - \lambda_n - \lambda_7 & \lambda_n & 0 & 0 & \lambda_7 \\ 0 & \mu_2 & 0 & 0 & -\mu_2 - \lambda_7 & 0 & 0 & \lambda_7 \\ 0 & 0 & \mu_1 & 0 & 0 & -\mu_1 - \lambda_n - \lambda_7 & \lambda_n & \lambda_7 \\ 0 & 0 & \mu_2 & 0 & 0 & 0 & -\mu_2 - \lambda_7 & \lambda_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where $A = \mu_1 + \lambda_n + \lambda_3 + \lambda_4 + F_1$ and $B = \mu_2 + \lambda_5 + \lambda_6 + F_2$.

From the matrix Q_c we get the group mean life time $MTTF_c$, which is used to calculate the overall server reliability. The state-space diagram for the server is the one of Figure 5.10(b), where the parameter λ takes the value $C \cdot \lambda_c$ and λ_c denotes the failure rate of one group with $\lambda_c = \frac{1}{MTTF_c}$. The server reliability is then calculated analogously to Eq. C.5.

Note that the example described in Figure 5.13 considers a small group size ($D_c = 4$). Increasing D_c increases the number of states contained in the state-space diagram of the group. In general, the number of states for a given D_c is: $1 + \sum_{i=0}^{\frac{D_c}{2}} 2^i = 2^{\frac{D_c}{2}+1}$. Below is a general method for building the state-space diagram of one group containing D_c disks for the Grouped One-to-One scheme:

We focus on the transitions from state $[i, j]$ to higher states and back. We know that state $[i, j]$

represents the total number $(i+j)$ of disks that have failed inside the group. These failures are due to i disk failures and j node failures. We also know that the number of states in the state-space diagram is $2^{\frac{D_c}{2}+1}$. The parameters i and j must respect the condition: $i+j \leq \frac{D_c}{2}$, since at most $\frac{D_c}{2}$ disk failures are tolerated inside one group. We distinguish between the two cases: (i) $i+j < \frac{D_c}{2}$ and (ii) $i+j = \frac{D_c}{2}$. Figure 5.14 shows the possible transitions and the corresponding rates for the case (i), whereas Figure 5.15 shows the transition for the case (ii).

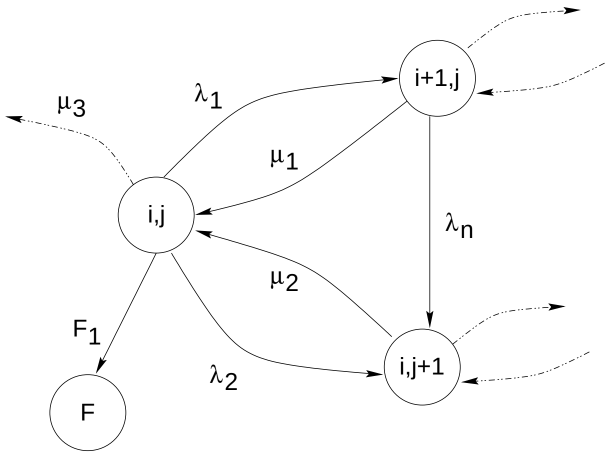


Figure 5.14: Transitions from state $[i,j]$ to higher states and back, for $(i+j) < \frac{D_c}{2}$.

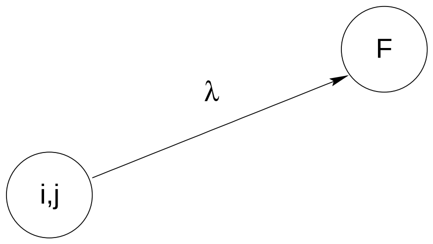


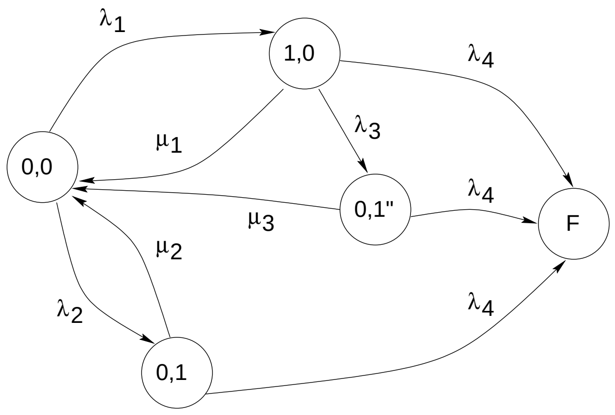
Figure 5.15: Transition to the failure state for $(i+j) = \frac{D_c}{2}$.

The parameters in the two figures 5.14 and 5.15 have the following values: $\lambda_1 = (D_c - (i+j)) \cdot \lambda_d \cdot P^{(i+j+1)}$, $\lambda_2 = (N-(i+j)) \cdot \lambda_n \cdot P^{(i+j+1)}$, $F_1 = ((D_c - (i+j)) \cdot \lambda_d + (N-(i+j)) \cdot \lambda_n) \cdot (1 - P^{(i+j+1)})$, $\mu_1 = \mu_d$, $\mu_2 = \mu_n$, $\mu_3 = (\mu_d \text{ OR } \mu_n)$, and $\lambda = (Dc - \frac{D_c}{2}) \cdot \lambda_d + (N - \frac{D_c}{2}) \cdot \lambda_n$.

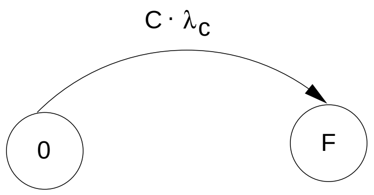
The One-to-Some Scheme

We use Orthogonal RAID for all One-to-Some schemes ($Mirr_{some-entire}$, $Mirr_{some-sub}$, and Par_{some}). If we consider again the data layouts of Figures 5.4, 5.5, and 5.7, Orthogonal RAID is then ensured if the following holds: Node 1 contains disks 0 and 3; node 2 contains disks 1 and 4; and node 3 contains disks 2 and 5.

For the reliability modeling of the One-to-Some scheme, we first build the state-space diagram for a single group (Figure 5.16(a)) and then compute the overall server reliability (Figure 5.16(b)). The states in Figure 5.16(a) denote the following: the initial state ($[0,0]$), the state where one disk fails ($[1,0]$), the state where one node fails resulting in a single disk failure within the group ($[0,1]$), the state where one disk and one node have failed and the failed disk belongs to the failed node ($[0,1”]$), and the one group failure state (F). The parameter values used in Figure 5.16 are: $\lambda_1 = D_c \cdot \lambda_d$, $\lambda_2 = N \cdot \lambda_n$, $\lambda_3 = \lambda_n$, $\lambda_4 = (D_c - 1) \cdot \lambda_d + (N-1) \cdot \lambda_n$, $\mu_1 = \mu_d$, $\mu_2 = \mu_n$, and $\mu_3 = min(\mu_d, \mu_n)$,



(a) State-space diagram for one group.



(b) State-space diagram for the server.

Figure 5.16: State-space diagrams for the One-to-Some Scheme for the case of dependent component failures.

The generator matrix Q_c for a group is:

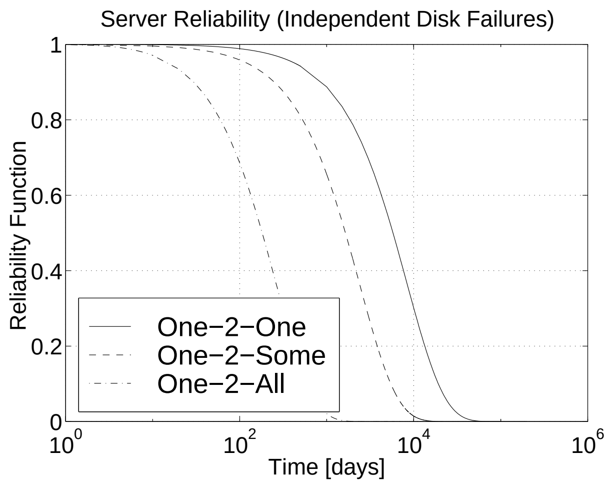
$$Q_c = \begin{pmatrix} -\lambda_1 - \lambda_2 & \lambda_1 & \lambda_2 & 0 & 0 \\ \mu_1 & -\mu_1 - \lambda_3 - \lambda_4 & 0 & \lambda_3 & \lambda_4 \\ \mu_2 & 0 & -\mu_2 - \lambda_4 & 0 & \lambda_4 \\ \mu_3 & 0 & 0 & -\mu_3 - \lambda_4 & \lambda_4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

5.4.4 Video Server Reliability: Results

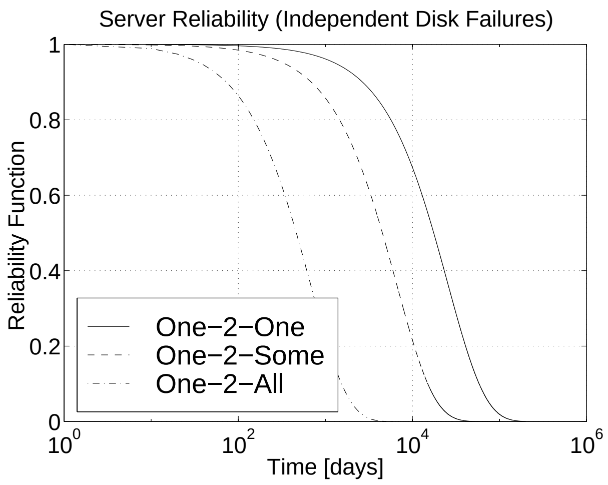
We resolve our continuous time Markov chains using the SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [SATR 96] tool for specifying and evaluating dependability and performance models. SHARPE takes as input the generator matrix and computes the server reliability at a certain time t.

Let us first consider the case of independent disk failures and let us take for the disk's failure rate λ_d the values $\lambda_d = \frac{1}{60000}\ \mathrm{hours}$ and $\lambda_d = \frac{1}{100000}\ \mathrm{hours}$ and for the disk's repair rate μ_d the value as $\mu_d = \frac{1}{72}\ \mathrm{hours}$ [2]. Further, let the total number of server disks considered is $D = 100$ and the number of server nodes is $N = 10$, each node containing 10 disks.

Figure 5.17 plots the server reliability for the One-to-One, One-to-All, and One-to-Some schemes for the case of independent disk failures using the above indicated parameter values. As expected, the server reliability for the One-to-One scheme is the highest. The One-to-Some scheme exhibits higher server reliability than the One-to-All scheme. Figures 5.17(a) and 5.17(b) also show how much the server reliability is improved when mean time to disk failure increases (λ_d decreases). For example, for the One-to-One scheme and after 10^4 days of operation, the server reliability is about 0.3 for $\lambda_d = \frac{1}{60000}\ \mathrm{hours}$ and is about 0.66 for $\lambda_d = \frac{1}{100000}\ \mathrm{hours}$.



(a) Server reliability for and $\lambda_d = \frac{1}{60000}\ \mathrm{hours}$.



(b) Server reliability for and $\lambda_d = \frac{1}{100000}\ \mathrm{hours}$.

Figure 5.17: Server reliability for the three schemes assuming independent disk failures with $\mu_d = \frac{1}{72}\ \mathrm{hours}$, $D = 100$, and $D_c = N = 10$.

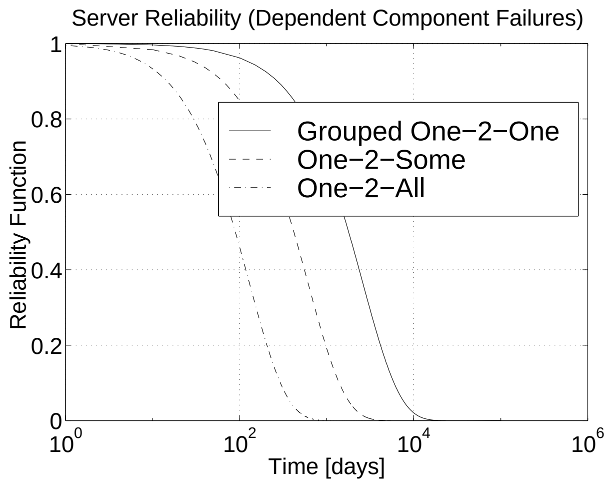
For the case of dependent component failures, we assume that $\lambda_d = \lambda_n$ and $\mu_d = \mu_n$. Let us first vary λ_d (λ_n) that will take the values $\frac{1}{60000}\ \mathrm{hours}$, $\frac{1}{100000}\ \mathrm{hours}$, and $\frac{1}{300000}\ \mathrm{hours}$, and keep μ_d (μ_n) constant as $\mu_d = \frac{1}{72}\ \mathrm{hours}$. Figure 5.18 depicts the server reliability for the Grouped One-to-One, the One-to-All, and the One-to-Some schemes for the case of dependent component

[2]The value $\mu_d = \frac{1}{72}\ \mathrm{hours}$ is a pessimistic value. We will later consider more optimistic values of this parameter.

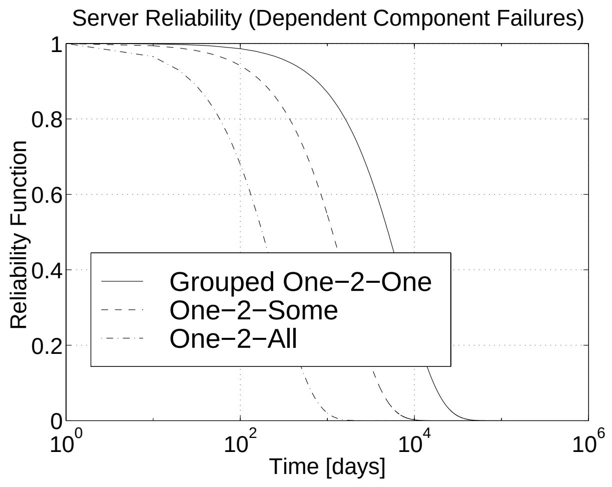
failures and based on the above parameter values. We observe that the Grouped One-to-One scheme provides a higher server reliability than the One-to-Some scheme. The One-to-All scheme has the lowest server reliability, e.g. for $\lambda_d = \lambda_n = \frac{1}{100000}$ hours and after three years of operation, the server reliability is 0 for the One-to-All scheme, 0.51 for the One-to-Some scheme, and 0.85 for the Grouped One-to-One scheme. Obviously, the server reliability increases as λ_d (λ_n) decreases, which can be seen in Figures 5.18(a), 5.18(b), and 5.18(c). Another important result is that the *reliability gap* between the Grouped One-to-One and One-to-Some schemes from one side and the One-to-All scheme from the other side increases as λ_d decreases.

We are now interested in studying the impact of varying the disk/node repair rate μ_d/μ_n on the server reliability for the different schemes considered. Figure 5.19(a) plots the server reliability for $\lambda_d = \lambda_n = \frac{1}{100000}$ hours and different values of μ_d/μ_n that are $\frac{1}{72}$ hours, $\frac{1}{48}$ hours, and $\frac{1}{3}$ hours. Figure 5.19(b) takes instead the value $\lambda_d = \lambda_n = \frac{1}{300000}$ hours for these various values of μ_d/μ_n. Obviously, as μ_d/μ_n increases, which implies that the mean time to repair decreases, the server reliability increases as well for all the three schemes considered. However, the increase in server reliability is more important for the One-to-Some scheme ($O2S$ with dashed lines) and the Grouped One-to-One scheme ($O2O$ with solid lines) than for the One-to-All scheme ($O2A$ with dash-dot lines). For the latter scheme, this increase becomes even invisible for a high value of λ_d/λ_n as observed in Figure 5.19(b) with $\lambda_d = \frac{1}{300000}$ hours. The results of Figure 5.19 show that an increase in the mean time to failure of disks and nodes (decrease in the disk/node failure rate λ_d/λ_n) and/or a decrease in their mean time to repair (increase in the disk/node repair rate μ_d/μ_n) clearly improve server reliability for both, the One-to-Some and the Grouped One-to-One scheme, whereas the One-to-All scheme is less sensitive to these changes and its server reliability remains *very low* even when we consider low values of λ_d and λ_n, i.e. $\frac{1}{300000}$ hours and high values of μ_d and μ_n, i.e. $\frac{1}{3}$ hours.

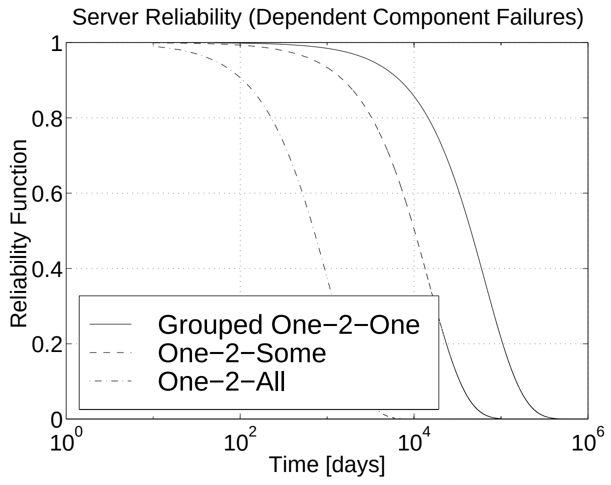
In summary, we have evaluated video server reliability for the three schemes One-to-All, One-to-Some, and Grouped One-to-One schemes when assuming both cases of independent disk failures and dependent component failures. We have considered various values of disk/node failure and repair rates and studied the impact of these parameters on the overall server reliability. The results of this section indicate that the Grouped One-to-One scheme always outperforms the other two schemes in terms of server reliability, whereas the One-to-All scheme always has the lowest server reliability. For the case of dependent component failures, we have also seen that the One-to-Some and the Grouped One-to-One schemes well benefit from the increase in the disk/node repair rate and/or the decrease in their failure rate, where the server reliability increases. The One-to-All scheme, however, remains insensitive to the increase of the disk/node repair rate or to the decrease of their failure rate. In the remainder of this chapter, we will restrict our reliability discussion to the case of dependent component failures, since it considers both, disk and node failures, and is therefore more realistic than the case of independent disk



(a) Server reliability for $\lambda_d = \lambda_n = \frac{1}{60000}$ hours.



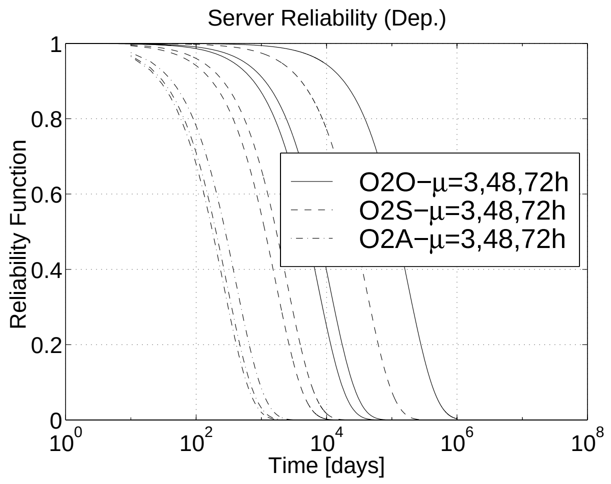
(b) Server reliability for $\lambda_d = \lambda_n = \frac{1}{100000}$ hours.



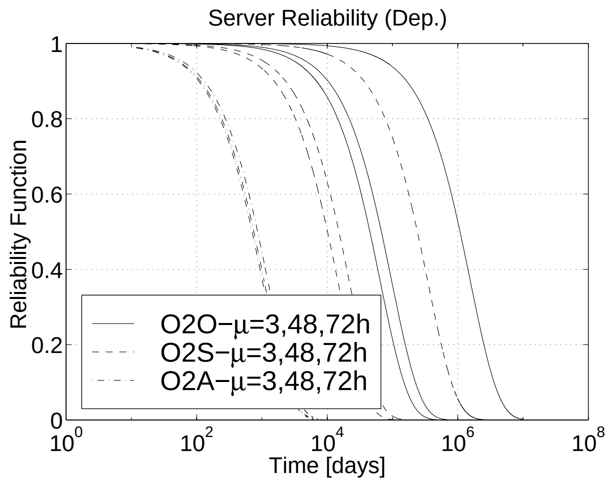
(c) Server reliability for $\lambda_d = \lambda_n = \frac{1}{300000}$ hours.

Figure 5.18: Server reliability for the three schemes assuming dependent component failures with $\mu_d = \frac{1}{72}$ hours, $D = 100$, and $D_c = N = 10$.

failures, where only disks are assumed to fail.



(a) Server reliability for $\lambda_d = \lambda_n = \frac{1}{100000}$ hours.



(b) Server reliability for $\lambda_d = \lambda_n = \frac{1}{300000}$ hours.

Figure 5.19: Server reliability for the three schemes assuming dependent component failures with $\mu_d = \frac{1}{72}, \frac{1}{48}$, and $\frac{1}{3}$ hours, $D = 100$, and $D_c = N = 10$.

5.5 Server Performance

In the last chapter, we have demonstrated that adding fault-tolerance within a server requires additional resources in terms of storage volume, main memory and I/O bandwidth capacity. We will see in this section that the reliability schemes considered differ not only in the throughput they achieve, but also in the amount of additional resources they need to guarantee uninterrupted service during failure mode. Server throughput[3] is therefore not enough to compare server performance of these schemes. Instead, we use the **cost per stream** as performance metric. We calculate in section 5.5.1 the server throughput for each of the schemes. Section 5.5.2 focuses on buffer requirements. Section 5.5.3 finally compares the different reliability schemes with respect to their per stream cost.

5.5.1 Server Throughput

The admission control policy decides, based on the remaining available resources, whether a new incoming stream is admitted. The CGS striping algorithm serves a list of streams from a single disk during one service round. During the next service round, this list of streams is shifted to the next disk. If Q_d denotes the maximum number of streams that a single disk can serve

[3]The server throughput is defined as the maximum number of streams that can be supported during failure mode and not during normal operation mode.

simultaneously (*disk throughput*) in a non fault-tolerant server, then the overall server throughput Q_s can be simply expressed as $Q_s = D \cdot Q_d$. Accordingly, we will restrict our discussion to disk throughput. Disk throughput Q_d is depicted in Eq. 2.1 of section 2.2 [GABI 98c]. The values of the different disk parameters used are those of Seagate and HP for the SCSI II disk drives [GKSZ 96] and are listed in Appendix A (Table A.1).

To allow for fault-tolerance, each disk reserves a portion of its available I/O bandwidth to be used during disk failure mode. Since the amount of reserved disk I/O bandwidth is not the same for all schemes, the disk throughput will also be different.

Let us start with the Grouped One-to-One scheme $Mirr_{one}$. Since the original content of a single disk is entirely replicated onto another disk, half of each disk's I/O bandwidth must be kept unused during normal operation mode to be available during disk failure mode. Consequently the disk throughput Q^{mirr}_{One} is simply the half of Q_d: $Q^{mirr}_{One} = \frac{Q_d}{2}$.

For the One-to-All mirroring scheme $Mirr_{all-entire}$ with entire block replication, the original blocks of one disk are spread among the other server disks. However, it may happen that the original blocks that would have been required from a failed disk during a particular service round are *all* replicated on the *same* disk (worst case situation). In order to guarantee deterministic service for this worst case, half of the disk I/O bandwidth must be reserved to disk failure mode. Therefore, the corresponding disk throughput $Q^{mirr}_{All-Entire}$ is: $Q^{mirr}_{All-Entire} = \frac{Q_d}{2}$.

The worst case retrieval pattern for the One-to-Some mirroring scheme $Mirr_{some-entire}$ with entire block replication is the same as for the previous scheme and we get: $Q^{mirr}_{Some-Entire} = \frac{Q_d}{2}$. Since the three schemes $Mirr_{one}$, $Mirr_{all-entire}$, and $Mirr_{some-entire}$ achieve the same throughput, we will use the term $Mirr_{Entire}$ to denote all of them and Q^{mirr}_{Entire} [4] to denote their disk throughput:

$$Q^{mirr}_{Entire} = \frac{Q_d}{2} \tag{5.6}$$

For the One-to-All mirroring scheme $Mirr_{all-sub}$ with sub-block replication, the situation changes. In fact, during disk failure mode, each disk retrieves at most $Q^{mirr}_{All-Sub}$ original blocks and $Q^{mirr}_{All-Sub}$ replicated *sub-blocks* during one service round. Let us assume that sub-blocks have the same size b^{all}_{sub}, i.e. $b = b^{all}_{sub} \cdot (D-1)$. The admission control formula becomes:

$$\begin{aligned} Q^{mirr}_{All-Sub} \cdot \left((\frac{b}{r_d} + t_{rot}) + (\frac{b^{all}_{sub}}{r_d} + t_{rot}) \right) + 2 \cdot t_{seek} \leq \tau \\ \Rightarrow \qquad Q^{mirr}_{All-Sub} = \frac{\tau - 2 \cdot t_{seek}}{\frac{b + b^{all}_{sub}}{r_d} + 2 \cdot t_{rot}} \end{aligned} \tag{5.7}$$

[4] These three schemes share the common property that each original block is entirely replicated into one block.

Similarly, the disk throughput $Q^{mirr}_{Some-Sub}$ for One-to-Some mirroring with sub-block replication $Mirr_{some-sub}$ is:

$$Q^{mirr}_{Some-Sub} = \frac{\tau - 2 \cdot t_{seek}}{\frac{b+b^{some}_{sub}}{r_d} + 2 \cdot t_{rot}} \tag{5.8}$$

where b^{some}_{sub} denotes the size of a sub-block as $b = b^{some}_{sub} \cdot (D_c - 1)$ and D_c is the number of disks contained on each group.

We now consider the disk throughput for the parity schemes. Recall that we study the buffering strategy and not the second read strategy for lost block reconstruction. For the One-to-All parity scheme Par_{all}, one parity block is needed for every $(D-1)$ original blocks. The additional load of each disk consisting in retrieving parity blocks when needed can be seen from Figure 5.6. In fact, for one stream in the worst case all requirements for parity blocks concern the same disk, which means that at most one parity block is retrieved from each disk every D service rounds. Consequently, each disk must reserve $\frac{1}{D}$ of its I/O bandwidth for disk failure mode. The disk throughput Q^{par}_{All} is then calculated as:

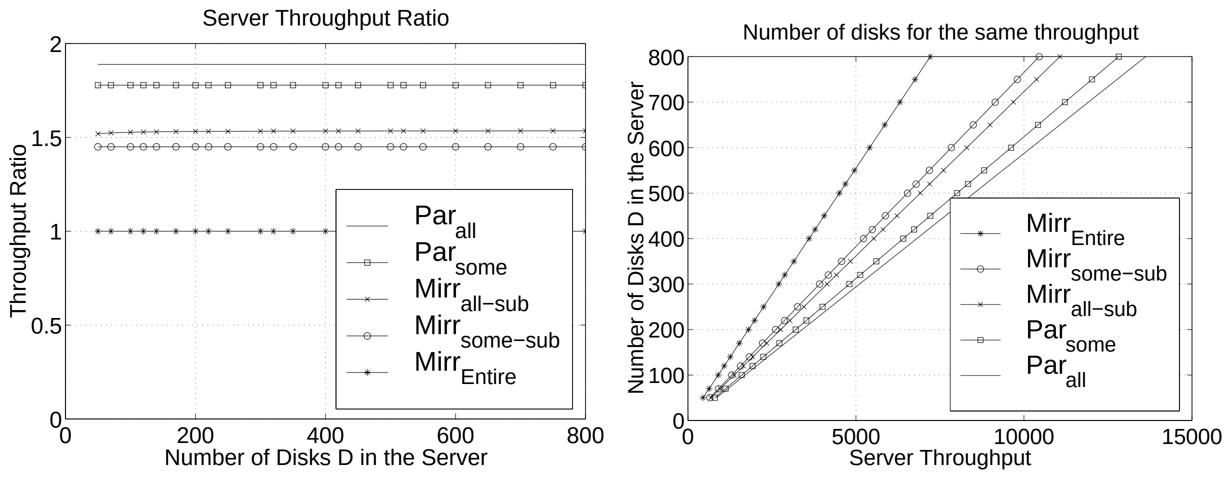
$$Q^{par}_{All} = Q_d - \lceil \frac{Q_d}{D} \rceil \tag{5.9}$$

Analogous to the One-to-All parity scheme, the One-to-Some parity scheme Par_{some} has the following disk throughput Q^{par}_{Some}:

$$Q^{par}_{Some} = Q_d - \lceil \frac{Q_d}{D_c} \rceil \tag{5.10}$$

In Figure 5.20(a), we take the throughput value Q^{mirr}_{Entire} of $Mirr_{Entire}$ as base line for comparison and plot the ratios of the server throughput as a function of the total number of disks in the server.

Mirroring schemes that use entire block replication ($Mirr_{Entire}$) provide lowest throughput. The two mirroring schemes $Mirr_{all-sub}$ and $Mirr_{some-sub}$ that use sub-block replication have throughput ratios of about 1.5. The performance for $Mirr_{all-sub}$ is slightly higher than the one for $Mirr_{some-sub}$ since the sub-block size $b^{all}_{sub} < b^{some}_{sub}$. Parity schemes achieve higher throughput ratios than mirroring schemes and the One-to-All parity scheme Par_{all} results in the highest throughput. The throughput for the One-to-Some parity scheme Par_{some} is slightly smaller than the throughput for Par_{all}. In fact, the parity group size of $(D-1)$ for Par_{all} id larger than D_c. As a consequence, the amount of disk I/O bandwidth that must be reserved for disk failure is smaller for Par_{all} than for Par_{some}. In order to get a quantitative view regarding the I/O bandwidth requirements, we reverse the axes of Figure 5.20(a) to obtain in Figure 5.20(b) for each scheme the number of disks needed to achieve a given server throughput.



(a) Throughput Ratios.

(b) Number of disks required for the same Server throughput.

Figure 5.20: Throughput results for the reliability schemes with $D_c = 10$.

5.5.2 Buffer Requirement

Another resource that affects the cost of the video server and therefore the cost per stream is main memory. We remind that for the SCAN scheduling algorithm, the worst case buffer requirement for one served stream is twice the block size b. Further, we consider dedicated buffer management that is to reserve for each stream separately of the other streams its worst case buffer requirement. During normal operation mode, the buffer requirement B of the server is therefore

$$B = 2 \cdot b \cdot Q_s$$

where Q_s denotes the server throughput. We calculate below the buffer requirement *during failure mode* for the different reliability schemes we consider.

Mirroring-based schemes replicate original blocks that belong to a single disk over one, all, or a set of disks. During disk failure mode, blocks that would have been retrieved from the failed disk are retrieved from the disks that store the replica. Thus, mirroring requires the same amount of buffer during normal operation mode and during component failure mode independently of the distribution granularity of replicated data. Therefore, for all mirroring schemes considered (Grouped One-to-One $Mirr_{one}$, One-to-All with entire block replication $Mirr_{all-entire}$, One-to-All with sub-block replication $Mirr_{all-sub}$, One-to-Some with entire block replication $Mirr_{some-entire}$, and One-to-Some with sub-block replication $Mirr_{some-sub}$) the buffer requirement during component failure is:

$$B^{mirr} = B$$

Unlike mirroring-based schemes, parity-based schemes need to perform a X-OR operation over a set of blocks to reconstruct a lost block. In fact, during normal operation mode the buffer is immediately liberated after consumption. When a disk fails, original blocks as well as the parity block that belong to the same parity group are sequentially retrieved (during consecutive service rounds) from consecutive disks and must be *temporarily stored* in the buffer for as many service rounds that elapse until the lost original block will be reconstructed [5]. Since buffer overflow must be avoided, the buffer requirement is calculated for the worst case situation where the whole parity group must be contained in the buffer before the lost block gets reconstructed. An additional buffer size of one block must be also reserved to store the first block of the next parity group. Consequently, during component failure, the buffer requirement B^{par}_{all} for Par_{all} is:

$$B^{par}_{all} = D \cdot b \cdot Q_s = \frac{D}{2} \cdot B$$

On the other side, the buffer requirement B^{par}_{some} for Par_{some} is:

$$B^{par}_{some} = D_c \cdot b \cdot Q_s = \frac{D_c}{2} \cdot B$$

Note that the buffer requirement for Par_{all} depends on D and therefore increases linearly with the number of disks in the server. For Par_{some}, however, the group size D_c can be kept constant while the total number of disks D varies. As a result, the buffer requirement B^{par}_{some} for Par_{some} remains unchanged when D increases.

5.5.3 Cost Comparison

The performance metric we use is the per stream cost. We first compute the total server cost $\$_{server}$ and then derive the cost per stream $\$_{stream}$ as:

$$\$_{stream} = \frac{\$_{server}}{Q_s}$$

We define the server cost as the cost of the hard disks and the main memory dimensioned for the component failure mode:

$$\$_{server} = P_{mem} \cdot B + P_d \cdot V_{disk} \cdot D$$

where P_{mem} is the price of $1\ \mathrm{Mbyte}$ of main memory, B the buffer requirement in Mbyte, P_d is the price of $1\ \mathrm{Mbyte}$ of hard disk, V_{disk} is the storage volume of a single disk in MByte,

[5]We remind that we consider the buffering strategy for a CGS striped video server.

and finally D is the total number of disks in the server. Current price figures – as of 1998 – are $P_{mem} = \$13$ and $P_d = \$0.5$. Since these prices change frequently, we will consider the relative costs by introducing the *cost ratio* α between P_{mem} and P_d: $\alpha = \frac{P_{mem}}{P_d}$. Thus, the server cost function becomes:

$$\$_{server} = P_{mem} \cdot B + \frac{P_{mem}}{\alpha} \cdot V_{disk} \cdot D = P_{mem} \cdot \left(B + \frac{V_{disk} \cdot D}{\alpha} \right)$$

Consequently, the per stream cost takes the formula of Eq. 5.11:

$$\$_{stream} = \frac{\$_{server}}{Q_s} = \frac{P_{mem} \cdot \left(B + \frac{V_{disk} \cdot D}{\alpha} \right)}{Q_s} \tag{5.11}$$

To evaluate the cost of the five different schemes, we compute for each scheme and for a given value of D the throughput Q_s achieved and the amount of buffer B required to support this throughput. Note that we take $D_c = 10$ for the schemes $Mirr_{some-entire}$, $Mirr_{some-sub}$, and Par_{some}.

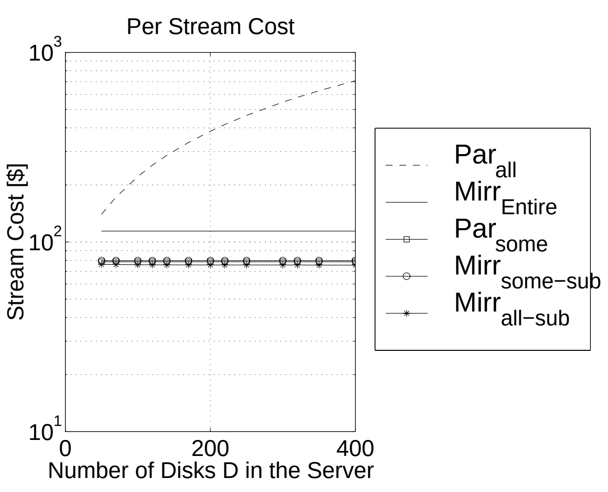
Figure 5.21 plots the per stream cost for the schemes Par_{all}, Par_{some}, $Mirr_{Entire}$, $Mirr_{some-sub}$, and $Mirr_{all-sub}$ for different values of the cost ratio α. We recall that the notation $Mirr_{Entire}$ includes the three mirroring schemes $Mirr_{all-entire}$, $Mirr_{some-entire}$, and the Grouped One-to-One $Mirr_{one}$ that experience the same throughput and require the same amount of resources. In Figure 5.21(a), we consider $\alpha = \frac{13}{0.5} = 26$ that presents the current memory/hard disk cost ratio. Increasing the value of $\alpha = \frac{P_{mem}}{P_d}$ means that the price for the disk storage drops faster than the price for main memory: In Figure 5.21(b), we multiply the current cost ratio by five to get $\alpha = 26 \cdot 5 = 130$ [6]. On the other hand, decreasing the value of α means that the price for main memory drops faster than the price for hard disk: In Figure 5.21(c) we divide the current cost ratio by five to get $\alpha = \frac{26}{5} = 5.2$ [7].

The results of Figure 5.21 indicate the following:

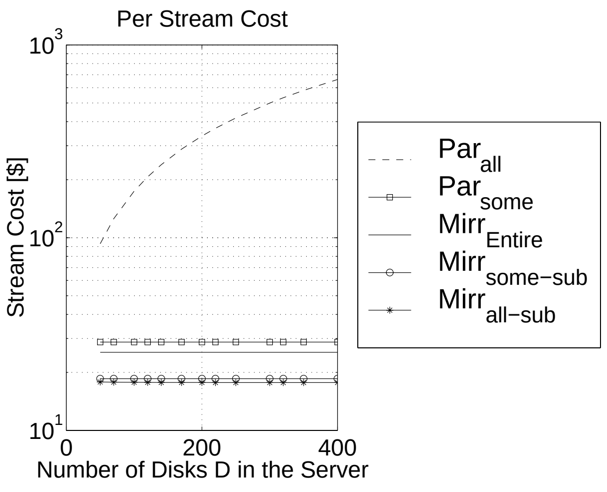
- The increase or the decrease in the value of α as defined above means a decrease in either the price for hard disk or the price for main memory respectively. Hence the overall decrease in the per stream cost in Figures 5.21(b) and 5.21(c) as compared to Figure 5.21(a).

[6] To illustrate the faster decrease of the price for hard disk as compared to the one for main memory, we consider the current price for main memory ($P_{mem} = \$13$) and calculate the new *reduced* price for hard disk ($P_d = \frac{13}{130} = \$0.1$).

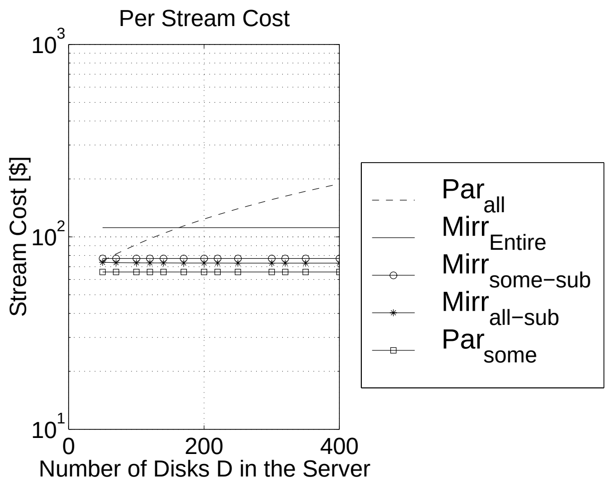
[7] Analogously, to illustrate the faster decrease of the price for memory as compared to the price for hard disk, we take the current price for hard disk ($P_d = \$0.5$) and calculate the new *reduced* price for memory ($P_{mem} = 0.5 \cdot 5.2 = \$2.6$).



(a) $\alpha = \frac{13}{0.5} = 26.$



(b) $\alpha = \frac{13}{0.1} = 130.$



(c) $\alpha = \frac{2.6}{0.5} = 5.2.$

Figure 5.21: Per stream cost for different values of the cost ratio α with $D_c = 10$.

- Figure 5.21(a) ($\alpha = 26$) shows that the One-to-All parity scheme (Par_{all}) results in the *highest* per stream cost that increases when D grows. In fact, the buffer requirement for Par_{all} is highest and also increases linearly with the number of disks D and thus resulting in the highest per stream cost. Mirroring schemes with entire block replication ($Mirr_{Entire}$) have the second worst per stream cost. The per stream cost for the remaining three schemes ($Mirr_{all-sub}$, $Mirr_{some-sub}$, and Par_{some}) is roughly equal and is *lowest*. The best scheme is the One-to-All mirroring scheme with sub-block replication

($Mirr_{all-sub}$). It has a slightly smaller per stream cost than the One-to-Some mirroring scheme with sub-block replication ($Mirr_{some-sub}$) due to the difference in size between b_{sub}^{all} and b_{sub}^{some} (see the explanation in section 6.4.1).

- The increase in the cost ratio α by a factor of five ($\alpha = 130$ in Figure 5.21(b)) slightly decreases the per stream cost of Par_{all} and results in a *dramatic* decrease in the per stream cost of all three mirroring schemes and also the parity scheme Par_{some}. For instance the per stream cost for Par_{some} decreases from $\$78.64$ down-to $\$28.72$ and the per stream cost of $Mirr_{some-sub}$ decreases from $\$79.79$ down-to $\$18.55$. All three mirroring schemes become more cost efficient than the two parity schemes.

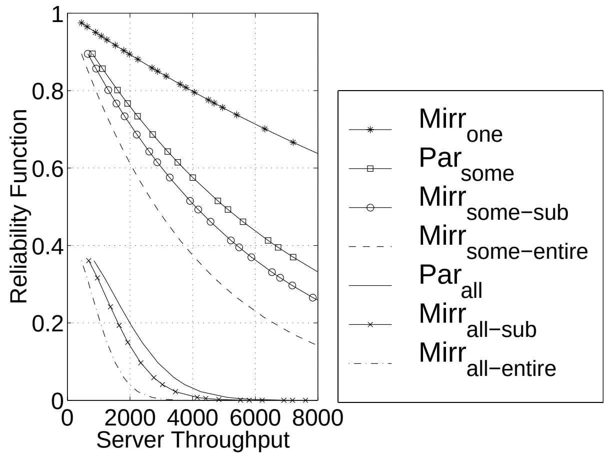
- The decrease in the cost ratio α by a factor of five ($\alpha = 5.2$ in Figure 5.21(c)) affects the cost of the three mirroring schemes very little. As an example, the per stream cost for $Mirr_{some-sub}$ is $\$79.79$ in Figure 5.21(a) and is $\$77.19$ in Figure 5.21(c). On the other hand, decreasing α, i.e. the price for main memory decreases faster than the price for hard disk, clearly affects the cost of the two parity schemes. In fact, Par_{some} becomes the most cost efficient scheme with a cost per stream of $\$65.64$. Although the per stream cost of Par_{all} decreases significantly with $\alpha = 5.2$, it still remains the most expensive for high values of D. Since Par_{all} has the highest per stream cost that linearly increases with D, we will not consider this scheme in further cost discussion.

5.6 Server Reliability vs. Server Performance

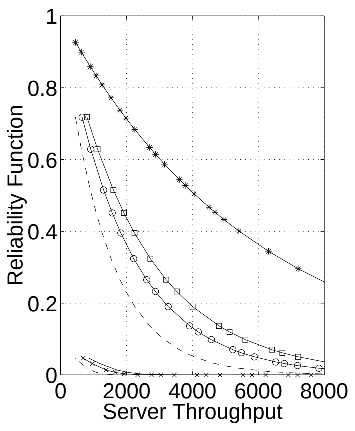
5.6.1 Server Reliability vs. Per Stream Cost

Figure 5.22 and Table 5.2 depict the server reliability and the per stream cost for the different reliability schemes discussed herein. The server reliability is computed after 1 year (Figure 5.22(a)) and after 3 years (Figure 5.22(b)) of server operation. Table 5.2 shows the normalized per stream cost for different values of α. We take the per stream cost of $Mirr_{one}$ as base line for comparison and divide the cost values for the other schemes by the cost for $Mirr_{one}$. We recall again that the three schemes $Mirr_{one}$, $Mirr_{all-entire}$ and $Mirr_{some-entire}$ have the same per stream cost since they achieve the same throughput given the same amount of resources (see section 5.5).

The three One-to-All schemes Par_{all}, $Mirr_{all-sub}$, and $Mirr_{all-entire}$ have poor server reliability even for a low values of server throughput, since they only survive a single disk failure. The difference in reliability between these schemes is due to the fact that Par_{all} requires, for the same throughput, fewer disks than $Mirr_{all-sub}$ that in turn needs fewer disks than $Mirr_{all-entire}$ (see Figure 5.20(b)). The server reliability of these three schemes decreases dramatically after



(a) Server reliability after 1 year of server operation.



(b) Server reliability after 3 years of server operation.

Figure 5.22: Server reliability for the same server throughput with $\lambda_d = \lambda_n = \frac{1}{100000}$ hours, $\mu_d = \mu_n = \frac{1}{72}$ hours, and $D_c = 10$.

	$\alpha = 26$	$\alpha = 130$	$\alpha = 5.2$
$Mirr_{one}$ $Mirr_{some-entire}$ $Mirr_{all-entire}$	1	1	1
Par_{some}	0.688	1.129	0.588
$Mirr_{some-sub}$	0.698	0.729	0.691
$Mirr_{all-sub}$	0.661	0.696	0.653

Table 5.2: Normalized stream cost (by $Mirr_{one}$) for different values of α with $D_c = 10$

three years of server operation as illustrated in Figure 5.22(b)). Accordingly, these schemes are not attractive to ensure fault tolerance in video servers and hence we are not going to discuss them more in the remainder of this paper. We further discuss the three One-to-Some schemes Par_{some}, $Mirr_{some-sub}$, and $Mirr_{some-entire}$ and the Grouped One-to-One scheme $Mirr_{one}$. Based on Figures 5.22(a) and 5.22(b), $Mirr_{one}$ has a higher server reliability than the three One-to-Some schemes Par_{some}, $Mirr_{some-sub}$, and $Mirr_{some-entire}$.

From Table 5.2, we see that $Mirr_{one}$, which has the same per stream cost as $Mirr_{some-entire}$, has a per stream cost about 1.5 higher than $Mirr_{some-sub}$. Par_{some} has the highest per stream cost for a high value of α ($\alpha = 130$) and is the most cost effective for a small value of α ($\alpha = 5.2$).

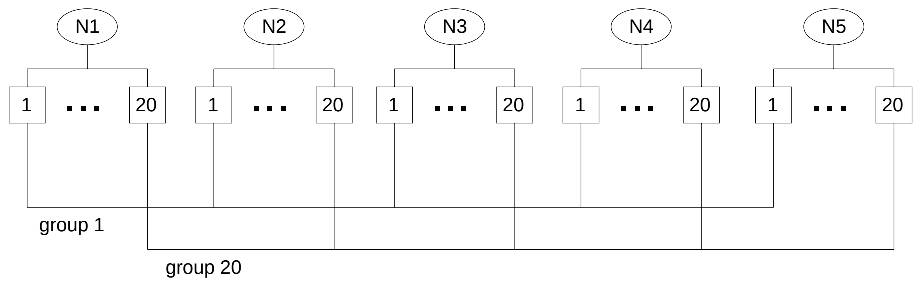
In conclusion, we see that the best scheme among the One-to-Some schemes is Par_{some} since it has a low per stream cost and requires fewer disks than $Mirr_{some-sub}$ and thus provides a higher server reliability than both, $Mirr_{some-sub}$ and $Mirr_{some-entire}$. Since $Mirr_{some-entire}$ achieves much lower server reliability than $Mirr_{one}$ for the same per stream cost, we conclude that $Mirr_{some-entire}$ is not a good scheme for achieving fault tolerance in a video server.

Based on the results of Figure 5.22 and Table 5.2, we conclude that the three schemes: $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$ are the good candidates to ensure fault tolerance in a video server. Note that we have assumed in Figure 5.22 for all these three schemes the same value $D_c = 10$. $Mirr_{one}$ has the highest server reliability but a higher per stream cost as compared to the per stream cost of Par_{some} and $Mirr_{some-sub}$. For the value $D_c = 10$, the two schemes Par_{some} and $Mirr_{some-sub}$ have a lower per stream cost but also a lower server reliability than $Mirr_{one}$. This difference in server reliability becomes more pronounced as the number of disks in the video server increases. We will see in the next section how to determine the parameter D_c for the schemes Par_{some} and $Mirr_{some-sub}$ in order to improve the trade-off between the server reliability and the cost per stream.

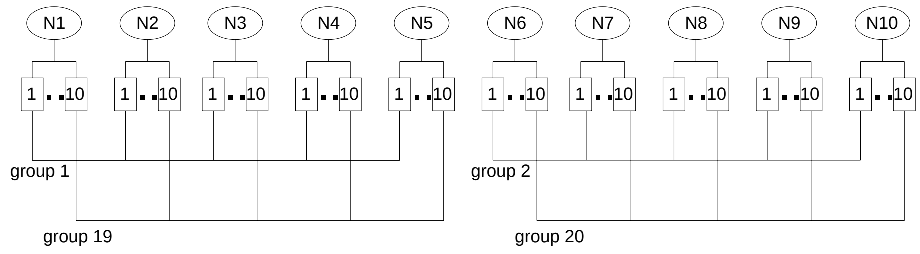
5.6.2 Determining the Group Size D_c

This section evaluates the impact of the group size D_c on the server reliability and the per stream cost. We limit our discussion to the three schemes: our $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$. Remember that we use the Orthogonal RAID principle to build the independent groups (see section 5.4.3). Accordingly, disks that belong to the same group are attached to different nodes. Until now, we have assumed that the group size D_c and the number of nodes N are constant ($D_c = N = 10$). In other terms, increasing D leads to an increase in the number of disks D_n per node. However, the maximum number of disks D_n is limited by the node's I/O capacity. Assume a video server with $D = 100$ disks and $D_c = 5$. We plot in Figure 5.23 two different ways to configure the video server. In Figure 5.23(a) the server contains five nodes ($N = 5$), where each node consists of $D_n = 20$ disks. One group contains $D_c = 5$ disks, each belonging to a different node. On the other hand, Figure 5.23(b) configures a video server with $N = 10$ nodes, each containing only $D_n = 10$ disks. The group size is again $D_c = 5$, i.e. a single group does not stretch across all nodes. Note that the number of groups C is the same for both configurations ($C = 20$). When the video server grows, the second alternative suggests to add new nodes (containing new disks) to the server, whereas the first alternative suggests to add new disks to the existing nodes. Since D_n must be kept under a certain limit given by the node's I/O capacity, we believe that the second alternative is more appropriate to configure a video server.

We consider two values the group size: $D_c = 5$ and $D_c = 20$ for the remaining three schemes $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$. Figures 5.24(a) and 5.24(b) depict the server reliability for $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$ after one year and after three years of server operation, respectively. Table 5.3 shows for these schemes the normalized per stream cost with different



(a) $D_c = N = 5$ and $D_n = 20$.



(b) $D_c = 5$, $N = 10$, and $D_n = 10$.

Figure 5.23: Video server configurations with $D = 100$ and $D_c = 5$.

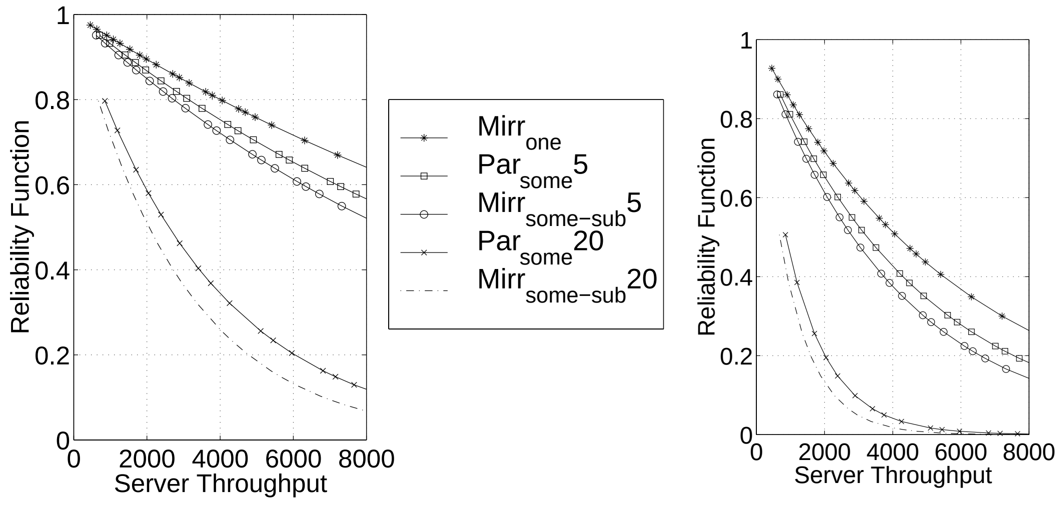
values of α and with $D_c = 5$ and $D_c = 20$. We take again the per stream cost of $Mirr_{one}$ as base line for comparison and divide the cost values for the other schemes by the cost for $Mirr_{one}$.

	D_c	$\alpha = 26$	$\alpha = 130$	$\alpha = 5.2$
$Mirr_{one}$		1	1	1
Par_{some}	20	0.798	1.739	0.584
Par_{some}	5	0.695	0.879	0.653
$Mirr_{some-sub}$	20	0.678	0.717	0.671
$Mirr_{some-sub}$	5	0.745	0.771	0.739

Table 5.3: Normalized stream cost (by $Mirr_{one}$) for different values of α and D_c.

The results of Figure 5.24 and Table 5.5.3 are summarized as follows:

- The server reliability of $Mirr_{one}$ is higher than for the other two schemes. As expected, the server reliability increases for both, Par_{some} and $Mirr_{some-sub}$ with decreasing D_c. In fact, as D_c decreases, the number of groups grows and thus the number of disk failures (one disk failure per group) that can be tolerated increases as well.



(a) Server reliability after 1 year of server operation.

(b) Server reliability after 3 years of server operation.

Figure 5.24: Server reliability for the same throughput with $\lambda_d = \lambda_n = \frac{1}{100000}$ hours, $\mu_d = \mu_n = \frac{1}{72}$ hours , and $D_c = 5, 20$..

- Depending on the value of α, the impact of varying the group size D_c on the per stream cost differs for Par_{some} and $Mirr_{some-sub}$. For $Mirr_{some-sub}$, the cost per stream decreases as the group size D_c grows for all three values of α considered. Indeed, the sub-block size to be read during disk failure is inversely proportional to the value of D_c. Consequently, the server throughput becomes smaller for decreasing D_c and the per stream cost increases. For Par_{some} with $\alpha = 26$ and $\alpha = 130$, the per stream cost decreases as D_c decreases. However, this result is reversed with $\alpha = 5.2$, where the per stream cost of Par_{some} is higher with $D_c = 5$ than with $D_c = 20$. The following explains the last observation:

  1. A small value of α (e.g. $\alpha = 5.2$) signifies that the price for main memory decreases faster than the one for hard disk and therefore main memory does not *significantly* affect the per stream cost for Par_{some} independently of the group size D_c.
  2. As the group size D_c decreases, the amount of I/O bandwidth that must be reserved on each disk for the disk failure mode increases. Consequently, the throughput is smaller with $D_c = 5$ than with $D_c = 20$. As a result, the per stream cost for Par_{some} increases when the group size D_c decreases.
  3. Since the memory cost affects *only little* the per stream cost of Par_{some} for a small value of α, the weight of the amount of I/O bandwidth to be reserved on the per stream cost becomes more *visible* and therefore the per stream cost of Par_{some} in-

creases as D_c decreases.

Note that the per stream cost of Par_{some} ($D_c = 20$) is *lowest* for $\alpha = 5.2$, whereas it is *highest* for $\alpha = 130$.

- Par_{some} has always a higher server reliability than $Mirr_{some-sub}$. Further, for high values of α, e.g. $\alpha = 130$, Par_{some} has a higher cost per stream than $Mirr_{some-sub}$ given the same value of D_c. However, for small values of α, e.g. $\alpha = 5.2$, Par_{some} becomes more cost effective than $Mirr_{some-sub}$.
- Based on the reliability results and for the high values of α, e.g. $\alpha = 26, 130$, we observe that a small group size ($D_c = 5$) considerably increases the server reliability and decreases the per stream cost for Par_{some}. For $Mirr_{some-sub}$, the server reliability increases as D_c decreases, but also the per stream cost slightly increases whatever the value of α is.

In summary, we have shown that the three schemes $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$ are good candidates to ensure fault-tolerance in a video server. The Grouped One-to-One scheme $Mirr_{one}$ achieves a higher reliability than the other two schemes at the expense of the per stream cost that is about 1.5 times as high. For Par_{some}, the value of D_c must be small to achieve a high server reliability and a low per stream cost. For $Mirr_{some-sub}$, the value of D_c must be small to achieve a high server reliability at the detriment of a *slight* increase in the per stream cost.

5.7 Summary

The goal of this chapter was to study reliability and performance of distributed and large video servers. The first step was to classify several reliability schemes. These schemes differ by the type of redundancy used (mirroring or parity) and by the distribution granularity of redundant data. We have retained seven reliability schemes and presented a data layout for each of them.

In order to quantitatively evaluate and compare these scheme, we have modeled server reliability using Continuous Time Markov Chains that were evaluated using the SHARPE software package. We have considered both cases: independent disk failures and dependent component failures. Furthermore, we have proposed a new mirroring-based scheme, called the Grouped One-to-One scheme that divides the video server into several independent groups similarly to the orthogonal RAID principle. Our results demonstrate that the Grouped One-to-One scheme outperforms all other schemes with respect to server reliability for both, independent disk failures and dependent component failures. The results further indicate that the One-to-All schemes, which only tolerate a single disk failure within the video server, have a *very low* server reliability that is not sufficient even for a small video server and dramatically decreases when the video server grows and is insensitive to the evolution of disk/node failure rate that is likely to decrease and disk repair rate that is likely to increase. Accordingly, the One-to-All schemes can not be used to ensure high reliability for large video servers.

A surprising, but very significant result of this chapter is that mirroring-based reliability, which doubles the storage volume required, can be more cost effective than parity-based reliability (see also [GAFS 99a, GAFS 99b]). Indeed, if we assume that hard disk prices will decrease faster than memory prices, mirroring-based schemes will continue to have a lower per stream cost than parity-based schemes. As an example, the $Mirr_{some-sub}$ mirroring scheme has a lower per stream cost than the Par_{some} parity scheme assuming the same group size. Moreover, if we add the processing costs to perform XOR decoding that are required for parity but not for mirroring, the cost advantage of mirroring-based reliability becomes more significant.

Finally, we have seen that out of the seven reliability schemes discussed in this chapter, only the $Mirr_{one}$, Par_{some}, and $Mirr_{some-sub}$ schemes achieve both, high server reliability and low per stream cost. We have compared these three schemes in terms of server reliability and per stream cost for several memory and hard disk prices and also for various group sizes. We found that the smaller the group size, the better the trade-off between high server reliability and low per stream cost [GABI 99b].

One of the major results of this chapter is that mirroring-based reliability is very *attractive* to ensure fault-tolerance for a video server. Besides the fact that mirroring is more cost effective than parity, the former significantly simplifies the design and implementation of a video server. In fact, mirroring does not require any synchronization of reads or additional processing time to decode lost blocks, which is needed for parity. Another advantage of mirroring is the disruption time after a disk failure. Indeed, mirroring takes at most one service round to send the replicated block expected. Parity, however, takes many service rounds to retrieve all blocks belonging to the parity group of the lost block and thus the disruption time will be higher. The next chapter exclusively studies mirroring-based reliability for a video server. We will propose a novel replica placement scheme that outperforms in terms of the overall server throughput the existing replica placement schemes such as the one used by the Microsoft Tiger [BOLO 96, BOLO 97] video server.

Chapter 6

ARPS: A Novel Replica Placement for Video Servers

6.1 Introduction

The last chapter has demonstrated that mirroring-based reliability is more attractive than parity-based reliability. Further, besides its cost effectiveness, mirroring significantly simplifies the design and the implementation of video servers, since in case of failure mirroring does not require any synchronization of reads or decoding to reconstruct the lost video data. Additionally, while mirroring doubles the amount of storage volume required, the steep decrease of the cost of magnetic disk storage, which is about 40% per year, makes mirroring more and more attractive as a reliability mechanism.

We remind that we are using as before the CGS striping algorithm to store/retrieve original blocks on/from disks. Now, we only focus on how to replicate original blocks of a single disk across the other disks of the server. Obviously, original blocks of one disk are not replicated on the same disk. Mirroring schemes differ on whether a single disk contains original and/or replicated data. The **mirrored declustering** scheme sees two *identical* disk arrays, where original content is replicated onto a distinct set of disks. When the server works in normal operation mode (disk failure free mode), only the half of the server disks are active, the other half remains idle, which results in load imbalances within the server.

Unlike mirrored declustering, **chained declustering** [HSDE 90, GOMZ 92] partitions each disk into two parts, the first part contains original blocks and the second part contains replicated blocks (copies): Original blocks of disk i are replicated on disk $(i+1)\mathrm{mod}D$, where D is the total number of disks of the server. **Interleaved declustering** is an extension of chained declustering, where original blocks of disk i are not entirely replicated on another disk $(i+1)\mathrm{mod}D$, but distributed over *multiple* disks of the server [MERC 95]. Note that chained declustering can be considered as a special case of interleaved declustering where the distribution granularity of replicated blocks equals 1.

We will restrict our discussion to interleaved declustering schemes, since these schemes distribute the total server load evenly among all components during normal operation mode. Note that interleaved declustering only indicates that the replica of the original blocks belonging to one disk are stored on *one*, *some*, or *all* remaining disks, but does not indicate *how* to replicate a *single* original block.

Mirroring is the conventional technique to ensure fault-tolerance in a disk-array based system. It corresponds to the RAID1 organization ([Patterson 88, LEED 93, CLGK 94]). Various video server designers [MOUR 96, BOLO 96, BOLO 97] have adopted mirroring to achieve fault-tolerance.

Mourad [MOUR 96] proposed the doubly striped scheme that is based on interleaved declustering, where original blocks of a disk are evenly distributed over *all* remaining disks of the sever. Thereby, an original block is entirely replicated on another disk. When a disk fails, it may happen with this scheme that all expected original blocks residing on the failed disk have their replica on the same disk (see the previous chapter). To ensure uninterrupted service for this worst case, half of the available bandwidth of each disk must be kept unused, which cuts the server throughput into the half. Another drawback of the doubly striped approach is that it does not tolerate more than one disk failure, which is not sufficient for large servers as our results have demonstrated in the previous chapter.

The Microsoft Tiger video server [BOLO 96] addresses the inefficiency of the doubly striped scheme and proposes to (i) divide the server into independent groups and (ii) perform sub-block replication instead of entire block replication. This scheme avoids to reserve the half of the available disk bandwidth for failure mode and tolerates a disk failure within each group. However, after a disk failure, the surviving disks of the group that contains that failed disk must perform twice as many read operations as compared to normal operation mode: The half of these read operations is for accessing original blocks, whereas the other half is for accessing replicated sub-blocks. Consequently, as the number of disk accesses doubles during disk failure mode, the number of seek operations doubles as well, which lowers the overall server throughput.

The main drawback of the two previous replication schemes is their additional *seek overhead* when operating in failure mode. In fact, these schemes require additional seek times to retrieve replicated data that are stored *separately* from original data. Unfortunately, high seek overhead decreases disk utilization and therefore server performance.

Instead, we present in this chapter a novel data layout strategy for replicated data on a video server, called ARPS (Adjacent Replica Placement Scheme). In contrast to classical replica placement schemes that store original and replicated data *not colocated* on a disk, ARPS stores replicated data *adjacent* to original data and thus does not require additional seek overhead when operating during failure mode. We will show that ARPS considerably improves the server performance compared to classical replica placement schemes such as the interleaved declustering scheme and the scheme used by the Tiger video server. Our performance metric is the server throughput that is, as defined in the previous chapters, the maximum number of users that the video server can support simultaneously.

The rest of this chapter is organized as follows. In section 6.2, we classify the interleaved declustering schemes for mirroring that might exist. We propose in section 6.3 a novel replica placement scheme that we call **ARPS** (**A**djacent **R**eplica **P**lacement **S**cheme). Section 6.4 compares ARPS with the classical interleaved declustering schemes in terms the overall server throughput. Finally, section 6.5 presents our conclusions.

6.2 Interleaved Declustering Schemes

This section briefly classifies different mirroring schemes for video servers. Note that we have already described most of the interleaved declustering schemes in the previous chapter. However, for sake of clarity, it is worthwhile to summarize these schemes as indicated below.

We present in Table 6.1 these interleaved declustering schemes, where we adopt two classification metrics: The first metric examines *how a single block is replicated*. The second metric concerns *the number of disks* that store the replica of the original content of a single disk.

We consider for the first metric the following three alternatives:

1. The copy of the original block is entirely stored on a single disk (One).
2. The copy of the original block is divided into a set of sub-blocks, which are distributed among some disks building an independent group (Some).
3. The copy of the original block is divided into exactly $(D-1)$ sub-blocks, which are distributed over all remaining $(D-1)$ server disks (All).

We distinguish three alternatives for the second metric:

1. The original blocks that are stored on one disk are replicated on a *single* disk (One).
2. The original blocks of one disk are replicated on a set of disks that build an independent group (Some).
3. The original blocks of one disk are replicated on the remaining $(D-1)$ server disks (All).

The symbol ”XXX” in Table 6.1 indicates combinations that are not useful for our discussion. The name of each scheme contains two parts. The first part indicates how an original block is replicated (the first metric) and the second part gives the number of disks, on which the content of one disk is distributed (the second metric). For instance, the scheme One/Some means that each original block is *entirely* replicated (One) and that the original content of one disk is distributed among a set of disks (Some).

	Single disk (One)	Set of disks (Some)	$(D-1)$ disks (All)
Entire block (One)	One/One	One/Some	One/All
Set of sub-blocks (Some)	XXX	Some/Some	XXX
$(D-1)$ sub-blocks (All)	XXX	XXX	All/All

Table 6.1: Classification of interleaved schemes

In the last chapter, we presented data layouts for each of the schemes presented in Table 6.1: see Figure 5.1 for One/One, 5.2 for One/All, 5.4 for One/Some, 5.5 for Some/Some, and finally Figure 5.3 for All/All.

Contrarily to the entire block replication organizations (One/One, One/Some, and One/All), the two sub-block replication organizations (Some/Some and All/All) avoid to reserve the half of each disk’s I/O bandwidth to ensure deterministic service during disk failure mode as we have seen in the last chapter. However, for both organizations, the number of seek operations will double during disk failure mode compared to normal operation mode. Exact values of the amount of I/O bandwidth to be reserved are given in section 6.4.1.

The main drawback of all replication schemes considered is their additional *seek overhead* when operating with disk failure as we will see in section 6.4.1. In fact, these schemes require additional seek times to retrieve replicated data that are stored separately from original data. Unfortunately, high seek overhead decreases disk utilization and therefore server throughput. We present in the following our novel scheme ARPS this problem by *eliminating* the additional seek overhead. In fact, we will see that our approach requires for the disk failure mode the same seek overhead as for the normal operation mode.

6.3 A Novel Replica Placement for Video Servers

6.3.1 Motivation

If we look at the evolution of SCSI disk’s performance, we observe that (i) data transfer rates double every 3 years, whereas (ii) disk access time decreases by one third every 10 years [Hennessy 90]. Figure 6.1 shows data transfer rates of different Seagate disks’ generations

(SCSI-I, SCSI-II, Ultra SCSI, and finally Ultra2 SCSI) [Mr.X]. Figure 6.2 depicts the evolution of *average* access time for Seagate disks. We see that Figures 6.1 and 6.2 confirm these observations.

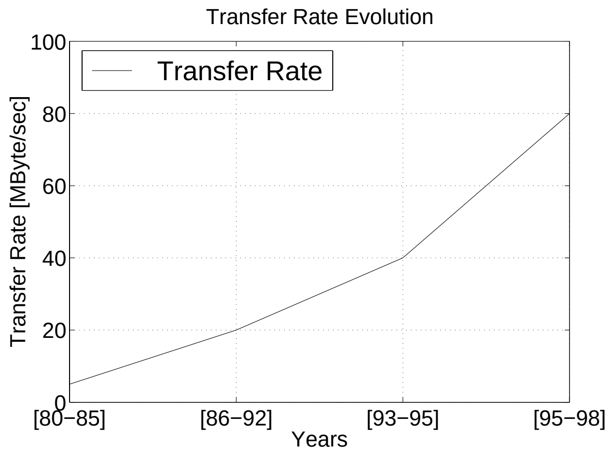


Figure 6.1: Evolution of data transfer rates for Seagate disks.

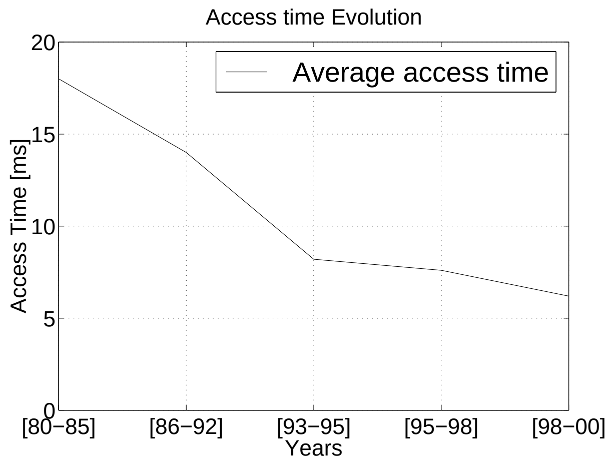


Figure 6.2: Evolution of average access time for Seagate disks.

We gave in chapter 2 an overview of the characteristics of magnetic disk drives. Thereby, we have concentrated on *seek time* and *rotational latency* that are the main parts of the seek operation when accessing a block stored on disk. We remind that seek time is in turn composed of four phases: (i) a *speedup* phase, which is the acceleration phase of the arm, (ii) a *coast* phase (only for long seeks), where the arm moves at its maximum velocity, (iii) a *slowdown* phase, which is the phase to rest close to the desired track, and finally (iv) a *settle* phase, where the disk controller adjusts the head to access the desired location. Note the duration t_{stl} of the the settle phase is independent of the distance traveled and is about $t_{stl} = 3 \ \text{ms}$. However, the durations of the speedup phase (t_{speed}), the coast phase (t_{coast}), and the slowdown phase ($t_{slowdown}$) mainly depend on the distance traveled. The seek time t_{seek} takes then the following form:

$$t_{seek} = t_{speed} + t_{coast} + t_{slowdown} + t_{stl}$$

Let us assume that the disk arm moves from the outer track (cylinder) to the inner track (cylinder) to retrieve data during one service round and in the opposite direction (from the inner track to the outer track) during the next service round (CSCAN). If a single disk can support up-to 20 streams, at most 20 blocks must be retrieved from disk during one service round. If we assume that the different 20 blocks expected to be retrieved are *uniformly* spread over the cylinders of the disk, we then deal only with *short* seeks and the coast phase is neglected (distance between two blocks to read is about 300 cylinders). Wilkes et al. have shown that seek time is a function of the distance traveled by the disk arm and have proposed for short seeks the formula $t_{seek} = 3.45 + 0.597 \cdot \sqrt{d}$, where d is the number of cylinders the disk arm must travel. Assuming that $d \approx 300$ cylinders, the seek time is then about $t_{seek} \approx 13.79 \ \mathrm{ms}$. Note that short seeks spend the most of their time in the speedup phase.

6.3.2 ARPS: Adjacent Replica Placement Scheme

The Some/Some scheme ensures a perfect distribution of the load of a failed disk over multiple disks and reduces the amount of bandwidth reserved for each stream on each surviving disk as compared to the interleaved declustering schemes (One/One, One/Some, and One/All). Since the content of one disk is replicated inside one group, Some/Some allows a disk failure inside each group. We call our approach, which is based on the Some/Some scheme, ARPS for Adjacent Replica Placement Scheme. The basic idea is to store original data as well as some replicated data in *a continuous* way so that when a disk fails, no additional seeks are performed to read the replica. In light of this fact, ARPS does not divide a disk in two *separate* parts, one for original blocks and the other for replicated blocks. Figure 6.3 shows an example of ARPS.

Let us consider only the content of group 1 (disks 1, 2, and 3) and the original block 9 that is stored on disk 3 (dashed block). The replication is performed as follows. We divide the original block into $3 - 1 = 2$[1] sub-blocks 9.1 and 9.2 that are stored *immediately* contiguous to the original blocks 7 and 8 respectively. Note that original blocks 7 and 8 represent the previous original blocks to block 9 within that group. If we take original block 13, its previous original blocks within that group are blocks 8 and 9. Now assume that disk 3 fails. Block 9 is reconstructed as follows. During the service round i where block 7 is retrieved, block 7 and sub-block 9.1 are *simultaneously* retrieved (neither additional seek time nor additional rotational latency, but additional read time). During the next service round $i + 1$, block 8 and sub-block

[1] 3 is the group size and therefore the number of sub-blocks is 2.

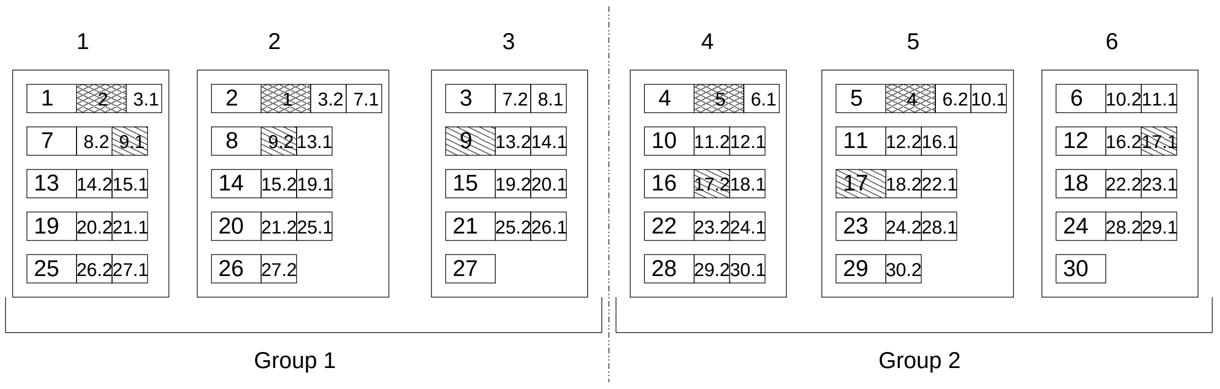


Figure 6.3: Layout example of ARPS for a server with 6 disks and 2 groups.

9.2 are simultaneously retrieved. Sub-blocks 9.1 and 9.2 are retrieved from server in advance and kept in buffer to be consumed during service round $i + 2$. Generally, sub-blocks that are read in advance are buffered for several service rounds before being consumed. The number of buffering rounds mainly depends on how large the server is (total number of server disks). If we assume that disk 1 is the failed disk, the reconstruction of block 19 is performed during the service rounds where blocks 14 (sub-block 19.1) and 15 (sub-block 19.2) are retrieved. The sub-blocks are kept in the buffer at most during 5 service rounds before they are consumed. The example shows that in order to simultaneously read one original block and one sub-block for one stream, data to be retrieved have a size of at most two original blocks. In order to ensure continuous read, one original block as well as the corresponding replicated blocks must be contained on the same track. Fortunately, today's disk drives satisfy this condition. In fact, the track size is continuously increasing. The actual mean track size for seagate new generation disk drives is about $160\ \mathrm{KByte}$, which is about $1.3\ \mathrm{Mbit}$. Hence the possibility to store inside *one* track the original block and the set of replicated sub-blocks as shown in Figure 6.3. ARPS therefore does not increase seek overhead, but *doubles*, in the worst case, the read time. Note that the very first blocks require special treatment: ARPS *entirely* replicates the two first blocks of a video within each group, which is represented in Figure 6.3 with the dark-dashed blocks (block 1 on disk 2, block 2 on disk 1 for group 1 and block 5 on disk 4, block 4 on disk 5 for group 2). Let us take the following example to explain the reason of doing this. If disk 1 has already failed before a new stream is admitted to consume the video presented in the figure, the stream is delayed for one service round. During the next service round, the two first blocks 1 and 2 are simultaneously retrieved from disk 2.

We present as follows a generalized data layout algorithm that implements ARPS. The parameters needed for this data layout algorithm are depicted in Table C.3.

Figure 6.4 illustrates the generalized data layout for ARPS. We explain below the meaning and values of the different parameters of Figure 6.4. The data layout formula depends on the disk

Term	Definition
D	The total number of server disks
D_c	The group size
C	The number of server groups
g_j	The j^{th} group in the server, with $j \in [1..C]$
$d_{i,j}$	the i^{th} disk in the j^{th} group, with $i \in [1..D_c]$ and $j \in [1..C]$

Table 6.2: Data layout parameters for ARPS

position within the group. We indentified three positions for a given group g_j.

The first position is the one of the first disk $d_{1,j}$ in the group. On this disk, original block b is stored adjacent to $D_c - 1$ replicated sub-blocks that are $b_1, \cdots, b_n$ with $b_1 = [b+1].[D_c - 1], \cdots, b_n = [b + D_c - 1].[1]$, where the notation $[\alpha].[\beta]$ identifies the β^{th} sub-block of the original block α.

The second position concerns all disks $d_{i,j}$ with $i \in [2..D_c - 1]$. On these disks, an original block k is stored adjacently to $D_c - 1$ replicated sub-blocks that are $k_1, \cdots, k_m$, $l_1, \cdots, l_p$ with $k_1 = [k+1].[D_c-1], \cdots, k_m = [k+D_c-i].[D_c-i], l_1 = [l].[D_c-(i-1)], \cdots, l_p = [l+(i-2)].[1]$ and $l = k + (D_c - i + 1) + (C - 1) \cdot D_c$.

The third position is the one of the last disk $d_{D_c,j}$ in the group. On this disk, an original block f is stored adjacent to $D_c - 1$ replicated sub-blocks that are $h_1, \cdots, h_q$ with $h_1 = [h].[D_c - 1], \cdots h_q = [h + D_c - 2].[1]$, and $h = f + (C - 1) \cdot D_c + 1$.

6.4 Performance Comparison

6.4.1 Admission Control Criterion

The admission control policy decides whether a new incoming stream can be admitted or not. The maximum number of streams Q that can be simultaneously admitted from server can be calculated in advance and is called server throughput. The server throughput depends on disk characteristics as well as on the striping/reliability scheme applied. In this paper, the difference between the schemes considered consists of the way original data is replicated. We consider the admission control criterion of Eq. 6.1. We first calculate disk throughput and then derive server throughput. Let Q_d denote the throughput achieved for a single disk. If we do not consider fault tolerance, the disk throughput is given in Eq. 6.1, where b is the block size, r_d is the data transfer rate of the disk, t_{rot} is the worst case rotational latency, t_{seek} is the worst case seek time,

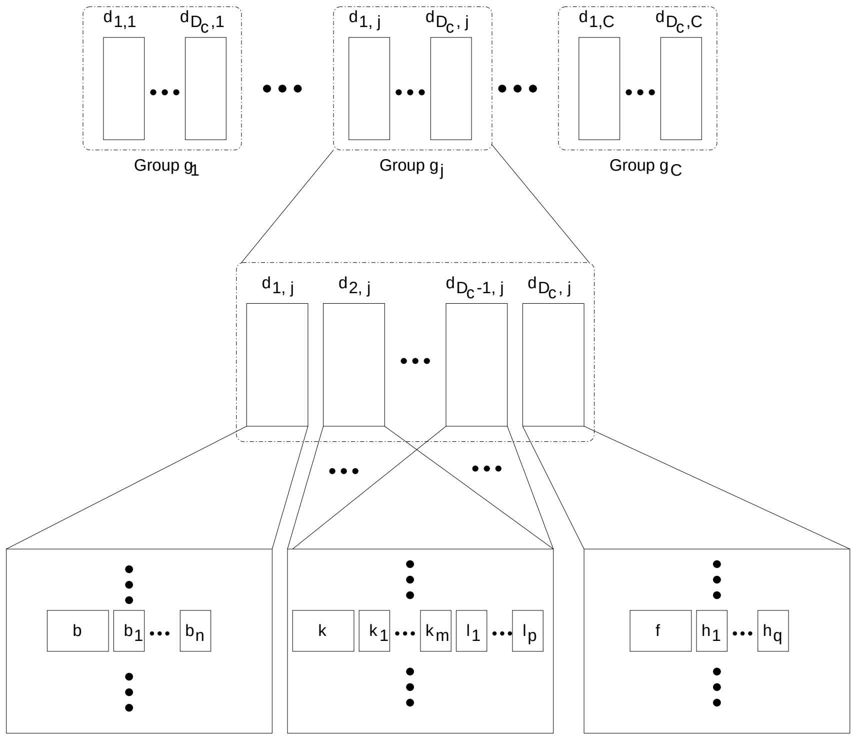


Figure 6.4: Data layout for ARPS.

and τ is the service round duration [2].

$$\begin{aligned} Q_d \cdot \left(\frac{b}{r_d} + t_{rot} + t_{seek} \right) &\leq \tau \\ Q_d &= \frac{\tau}{\frac{b}{r_d} + t_{rot} + t_{seek}} \end{aligned} \tag{6.1}$$

Introducing fault tolerance (mirroring-based), the disk throughput changes and becomes dependent on which mirroring scheme is applied. Three schemes are considered for discussion: the One/Some scheme, the Microsoft Some/Some scheme, and our ARPS. Let Q_d^{OS}, Q_d^{SS}, and Q_d^{ARPS} the disk throughput for One/Some, Some/Some, and our ARPS, respectively. Note that the disk throughput is the same during both, normal operation and disk failure mode.

For the One/Some scheme, half of the disk I/O bandwidth should be reserved in the worst case to reconstruct failed original blocks and thus Q_d^{OS} is calculated following Eq. 6.2.

[2]We take a constant value of τ, typically $\tau = \frac{b}{r_p}$, where b is the size of an original block and r_p is the playback rate of a video

$$Q_d^{OS} = \frac{Q_d}{2} = \frac{\left(\frac{\tau}{\frac{b}{r_d}+t_{rot}+t_{seek}}\right)}{2} \tag{6.2}$$

For the Some/Some scheme, in order to reconstruct a failed original block, the retrieval of sub-blocks requires small read overhead (small sub-blocks to read on each disk), but a *complete* latency overhead for each additional sub-block to read from disk. The admission control criterion presented in Eq. 6.1 is therefore modified as Eq. 6.3 shows. The parameter b_{sub} denotes the size of a sub-block.

$$\begin{aligned} Q_d^{SS} \cdot \left((\frac{b}{r_d} + t_{rot} + t_{seek}) + (\frac{b_{sub}}{r_d} + t_{rot} + t_{seek}) \right) \leq \tau \\ Q_d^{SS} = \frac{\tau}{\frac{b+b_{sub}}{r_d} + 2 \cdot (t_{rot} + t_{seek})} \end{aligned} \tag{6.3}$$

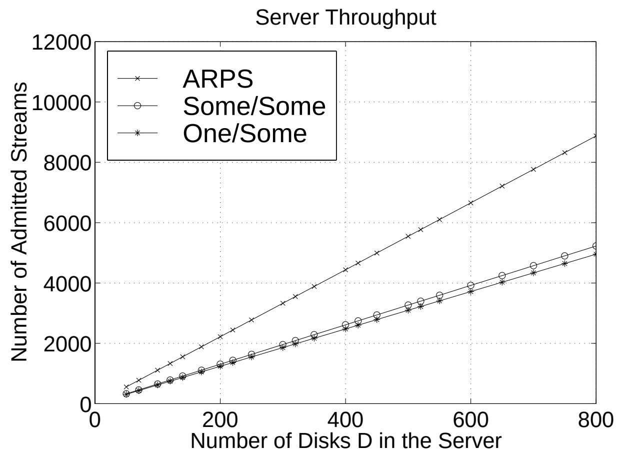
Let us have a look at ARPS and consider the case where a disk fails inside one group. The following criterion holds for admission control (Eq. 6.4), where b_{over} denotes the amount of data (overhead) that should be simultaneously read with each original block. In the worst case $b_{over} = b$.

$$\begin{aligned} Q_d^{ARPS} \cdot \left(\frac{b + b_{over}}{r_d} + t_{rot} + t_{seek} \right) \leq \tau \\ Q_d^{ARPS} = \frac{\tau}{\frac{2 \cdot b}{r_d} + t_{rot} + t_{seek}} \end{aligned} \tag{6.4}$$

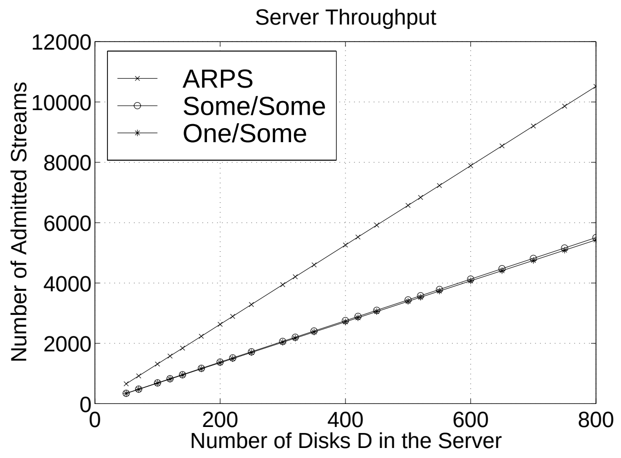
Once the disk throughput Q_d is calculated, the server throughput Q can be easily derived as $Q = D \cdot Q_d$ for each of the schemes, where D denotes again the total number of disks on the server.

6.4.2 Throughput Results

We present in the following the results of the server throughput Q^{OS}, Q^{SS}, and Q^{ARPS} respectively for the schemes One/Some, Some/Some, and ARPS. In Figure 6.5, we keep constant the values of the seek time and the rotational latency and vary data transfer rate r_d of the disk. Figures 6.5(a) and 6.5(b) show that ARPS outperforms the Microsoft Some/Some scheme that itself outperforms One/Some for all values of r_d ($20, 80 \ \mathrm{MByte/sec}$). Figure 6.5 also shows



(a) Throughput for $r_d = 20\ \mathrm{MByte/sec}$.



(b) Throughput for $r_d = 80\ \mathrm{MByte/sec}$.

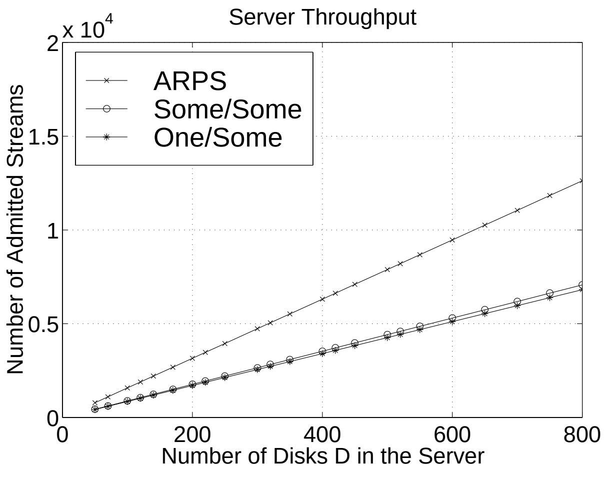
Figure 6.5: Server throughput for One/Some, Some/Some, and ARPS with $t_{seek} = 13.79\ \mathrm{ms}$, $t_{rot} = 10\ \mathrm{ms}$, $b = 0.5\ \mathrm{Mbit}$, and $r_p = 1.5\ \mathrm{Mbit/sec}$.

	$\frac{Q^{ARPS}}{Q^{OS}}$	$\frac{Q^{ARPS}}{Q^{SS}}$
$r_d = 20\ \mathrm{MByte/sec}$	1.79	1.69
$r_d = 40\ \mathrm{MByte/sec}$	1.88	1.83
$r_d = 80\ \mathrm{MByte/sec}$	1.93	1.91

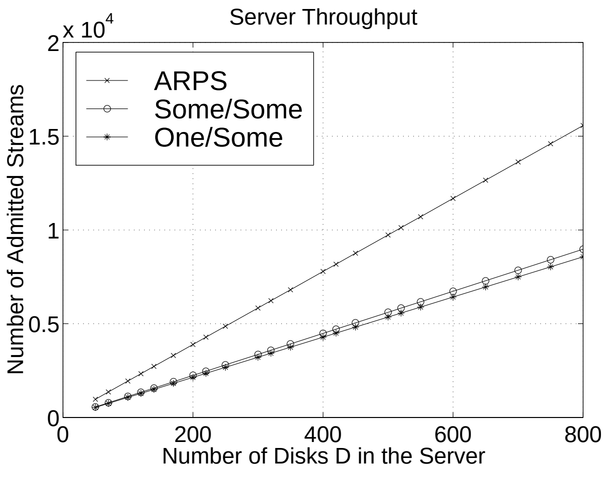
Table 6.3: Throughput ratios.

that the gap between ARPS and the two other schemes (One/Some and Some/Some) *considerably* increases with the increase of the data transfer rate r_d. Table 6.3 illustrates the benefit of ARPS, where the ratios $\frac{Q^{ARPS}}{Q^{OS}}$ and $\frac{Q^{ARPS}}{Q^{SS}}$ are illustrated depending on r_d.

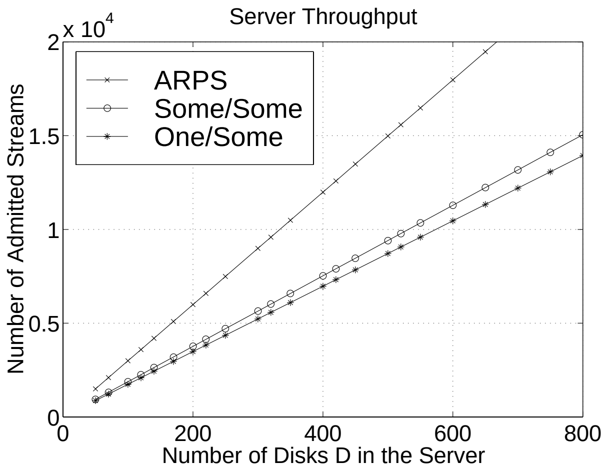
We focus now on the impact of the evolution of the seek time t_{seek} and the rotational latency t_{rot} on the throughput for the three schemes considered. We keep constant the data transfer rate that takes a relatively small value of r_d ($40\ \mathrm{Mbyte/sec}$). Figure 6.6 plots server throughput for the corresponding parameter values. The Figure shows that ARPS achieves *highest* server throughput for *all* seek time/rotational latency combination values adopted. Obviously, the decrease in t_{seek} and t_{rot} increases throughput for all schemes considered. We notice that the gap between our ARPS and the Some/Some *slightly* decreases when t_{seek} and t_{rot} decrease as Table 6.4 depicts, where disk transfer rate has also the value r_d ($40\ \mathrm{Mbyte/sec}$). Note that th value $t_{rot} = 6\ \mathrm{ms}$ corresponds to a spindle speed of $10000\ \mathrm{prm}$, and the value $t_{rot} = 4\ \mathrm{ms}$ corresponds to the speed of $15000\ \mathrm{prm}$, which is a too optimistic value. We observe that even for very low values of t_{seek} and t_{rot}, ARPS outperforms the Tiger Some/Some in terms of server throughput ($\frac{Q^{ARPS}}{Q^{SS}} = 1.59$).



(a) Throughput for $t_{seek} = 10$ ms, $t_{rot} = 8$ ms.



(b) Throughput for $t_{seek} = 8$ ms, $t_{rot} = 6$ ms.



(c) Throughput for $t_{seek} = 4$ ms, $t_{rot} = 4$ ms.

Figure 6.6: Server throughput for different access time values with $r_d = 40$ MByte/sec, $b = 0.5$ Mbit, and $r_p = 1.5$ Mbit/sec.

6.4.3 Reducing Read Overhead for ARPS

The worst case read overhead of ARPS is the time to read redundant data of the size of a *complete* original block. We present in the following a method that reduces this worst case amount

	$\frac{Q^{ARPS}}{Q^{SS}}$
$t_{seek} = 13,79$ and $t_{rot} = 10$	1.83
$t_{seek} = 10$ and $t_{rot} = 8$	1.78
$t_{seek} = 8$ and $t_{rot} = 6$	1.73
$t_{seek} = 4$ and $t_{rot} = 4$	1.59

Table 6.4: Throughput ratio between ARPS and the Tiger Some/Some.

of data read down-to the half of the size of one original block. This method simply consists of storing different sub-blocks not only on one side of one original block, but to distribute them on the left as well as on the right side of the original block. Figure 6.7 shows an example, where each original block is stored in the middle of two replicated sub-blocks. Let us assume that disk 3 fails and that block 9 must be regenerated. While reading block 7, disk 1 continuous its read process and reads sub-block 9.1. On disk 2, the situation is slightly different. In fact, before reading block 8, sub-block 9.2 is read. In this particular example, no useless data is read, in contrast to the example in Figure 6.3.

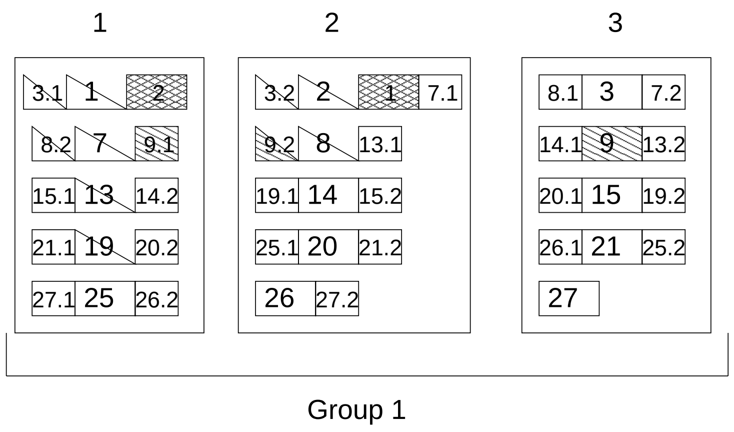


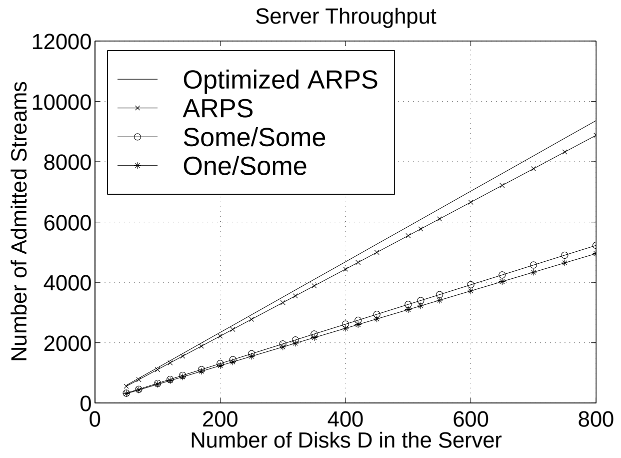
Figure 6.7: Reducing read overhead for ARPS.

Optimizing the placement of sub-blocks as presented in Figure 6.7 reduces the worst case read overhead to $\frac{b}{2}$ for disk failure mode. Accordingly, the admission control formula follows Eq. 6.5. Let us call this new method the **Optimized ARPS** scheme and its disk throughput Q_d^{OARPS}.

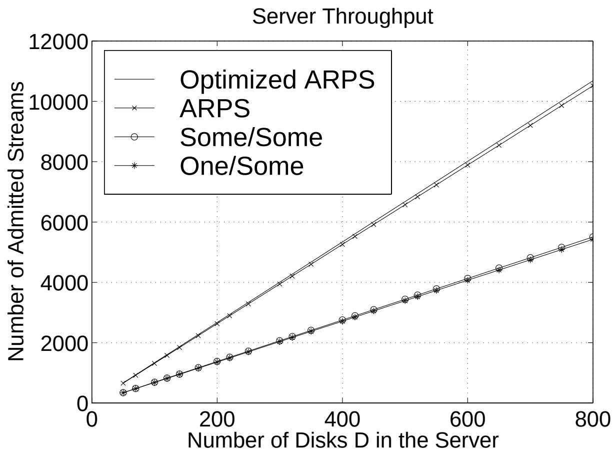
$$Q_d^{OARPS} = \frac{\tau}{\frac{\frac{3}{2} \cdot b}{r_d} + t_{rot} + t_{seek}} \tag{6.5}$$

Figure 6.8 plots the server throughput results of the One/Some, Some/Some, ARPS, and Optimized ARPS for different values of data transfer rate r_d. We observe that Optimized

ARPS *slightly* improves server throughput as compared to ARPS. The ratio decreases as disk transfer rate increases. In fact we notice a 10% improvement in server throughput for $r_d = 20\ \mathrm{MByte/sec}$, 5% improvement for $r_d = 40\ \mathrm{MByte/sec}$, and only 2% improvement for $r_d = 80\ \mathrm{MByte/sec}$.



(a) Throughput for $r_d = 20\ \mathrm{MByte/sec}$.



(b) Throughput for $r_d = 80\ \mathrm{MByte/sec}$.

Figure 6.8: Server throughput for One/Some, Some/Some, ARPS, and Optimized ARPS with $t_{seek} = 13.79\ \mathrm{ms}$, $t_{rot} = 10\ \mathrm{ms}$, $b = 0.5\ \mathrm{Mbit}$, and $r_p = 1.5\ \mathrm{Mbit/sec}$..

6.5 Summary

In this chapter, we have proposed a novel data replication strategy for video servers, called ARPS for Adjacent Replica Placement Scheme. In contrast to the classical replica placement schemes, where each single disk of the server is dedicated to original video data and redundancy data, ARPS splices replicated data and original data and thus does not require any additional seek time and rotational latency when operating in the presence of disk failures. The results show that ARPS achieves 60% to 90% more server throughput than classical interleaved declustering schemes like the one proposed for the Microsoft Tiger video server. Further, we have seen that ARPS, for pessimistic as well as for optimistic values of disk transfer rates and disk rotational and seek times, always achieves highest throughput as compared to the classical data replication schemes. Finally, we have optimized ARPS to reduce read overhead and noticed a slight increase in the server throughput [GABI 99a].

ARPS, however, requires an additional processing overhead to treat the different sub-blocks retrieved with each original block during disk failure mode. This additional processing overhead consists of the capture of the expected sub-blocks that must reconstruct the lost original block. These sub-blocks have to be temporarily stored until they are needed for display during a future service round. In order to keep the buffer management as simple as possible at the server side, we propose to shift the additional overhead to the client. Since one disk failure is allowed within each group, the video server is able to tolerate at most C disk failures. Thus, the client must reserve C buffers, each of them having the size b of an original block. Whenever the client receives an original block followed by $D_c - 1$ replicated sub-blocks, it passes the original block to the display entity and copies the needed replicated sub-block into ots allocated buffer. Obviously, ARPS ensures that lost blocks are reconstructed well before their deadlines. If we consider the Optimized ARPS scheme for a group size of 3, during each round, the client receives the expected original block followed by only *one* replicated sub-block, which is the one of the lost orginal block due to a disk failure within that group.

Chapter 7

Implementation of Eurecom's Video Server Prototype

7.1 Motivation

Video servers have completely different requirements than classical file systems. In fact, a video server must store and retrieve video streams that have special constraints: The data intensive nature of video streams requires large amount of storage and the bandwidth intensive nature demands high I/O bandwidth at the disk level, but also at the network level. Further, the periodic and real-time nature of video streams require mechanisms such as stream scheduling and admission control to provide guaranteed stream delivery from server to clients. Additionally, standard fault-tolerance schemes, e.g. the hardware RAID organizations, that were conceived for classical disk arrays are not adapted to the video server requirements. Hence the need for video server reliability to conceive adequate schemes.

Throughout the previous chapters, we have concentrated on video server design and performance. The primary criteria we have considered are (i) **high server throughput** that is the maximum number of streams that can be serviced concurrently, (ii) **scalability and cost effectiveness** that enables the video server to be economically viable, and (iii) **guaranteed quality of service** that include continuous and uninterrupted service that holds even in the presence of failures (server reliability), load-balancing, and start-up latency. One of our challenges is to build a *complete video server design* that addresses all of the different design phases and collectively satisfies the video server requirements and the criteria (i), (ii), and (iii). Based on this design, this chapter investigates video server implementation and describes the video server prototype we have developed at Eurecom.

If we try to combine the two criteria (i) and (ii) cited above, we are conducted to formulate the following challenge: *The video server must achieve high throughput at low costs*. Hence the motivation to use standard hardware (workstations and PCs) to implement the video server. Besides the price advantage, a video server based on standard hardware can continuously profit from the technological advances of these components. However, using cheap hardware requires to shift the video server complexity to the application and therefore to realize the whole intelligence of the server in software. The aim of the last chapters was to design such a video server. Thereby, we have made various design decisions that we will adopt for the server implementation. These decisions mainly include the following:

- *The server array*: In order to ensure scalability and high sever throughput, our video server architecture is based on the *server array* that contains multiple server nodes, each of them is connected to multiple magnetic disks. The different nodes of the server array are connected via two networks that can be used alternatively: Ethernet allowing $10 \mathrm{Mbit/sec}$ traffic and ATM switch allowing $155 \mathrm{Mbit/sec}$ traffic.

- *Round-based retrieval and SCAN stream scheduling*: The SCAN stream scheduling algorithm is used to optimize the disk utilization, whereas data retrieval is based on equal size service rounds.

- *CGS*: For data striping, the CGS algorithm is used to ensure high server throughput and low buffer requirement.

- *Mirroring*: Server reliability is ensured by mirroring. In order to tolerate even a whole node failure, the original content of one node is *entirely* replicated on another node of the server array.

The rest of this chapter is organized as follows. The next section presents related work. In section 7.3, we describe the different components of the video server prototype that are the meta server, the disk servers, and the clients. Section 7.4 addresses client-server interaction and presents the steps to take from the client request until the video playback at the client side. In section 7.5, we show how our server prototype operates in failure mode. Section 7.6 evaluates the performance of the video server prototype and finally, section 7.7 summarizes our results.

7.2 Related Work

Various video server implementations have been realized in the past few years, either for commercial purpose or in the context of research prototypes. Several major companies in computer systems business (IBM, SGI, HP) and computer software (Oracle, Microsoft, IBM) have developed their commercial video servers. Except the Microsoft Tiger video server, all of these products, however, rely on powerful parallel machines such as the Oracle NCube server solution. These products provide *huge* bandwidth capacity and are designed to service thousands of clients concurrently. In spite of their high throughput promise, these products are *very expensive* and are therefore *not* economically viable.

Instead, other companies have developed video servers based on standard and cheap hardware such as the Tiger video server [BOLO 96, BOLO 97] of Microsoft and the StarWorks video server [TOB 93b] of Starlight Networks. Whereas the latter is designed to support only up-to hundred clients simultaneously and is based on the hardware RAID3 parity-based architecture, the former is based on mirroring-based reliability and implements a scalable architecture that is able to serve thousands of clients.

Several video server prototypes have been implemented. Many of these prototypes are based on standard (hardware) RAID architectures [BUD 94, BUD 95, BUDD 96]. Other prototypes are based on software RAID-like solutions such as the Mitra server of the University Of Southern California [Ghandeharizadeh 98] and the Fellini server of Bell Labs [Cliff 96]. Mitra is a multimedia server that is able to deal with heterogeneous magnetic disks regarding storage capacity and disk transfer rate. It further implements GSS for stream scheduling and is based on a similar striping algorithm as MGS, where a configuration planer determines the striping granularity of the video objects to store. Fellini is conceived to support multimedia data and classical data as well. It further implements the CGS algorithm for data striping and uses a deterministic admission control criterion, which is similar to ours. Further, Fellini applies a RAID5-like solution to be fault-tolerant.

We have implemented a video server prototype based on our results presented in previous chapters. Implementation challenges were to (i) develop *multi-platform* video server, where the server as well as the clients run on both, UNIX machines (Solaris) and PCs (Windows NT, Windows 95), (ii) perform distributed scheduling and data delivery that allow for server scalability, and (iii) ensure fault-tolerance, where disk/node failure detection and data recovery are performed at the application level. The implementation of the video server prototype is the subject of an industrial project with CNET/France Telecom. The rest of this chapter describes the implementation of our prototype in more details.

7.3 Video Server Components

Figure 7.1 presents the overall video server architecture. It mainly contains three components: (i) a central entity, called **meta server**, (ii) a set of server nodes, also called **disk servers**, and (iii) multiple video server **clients**. We describe as follows functionality and implementation aspects for each of these components.

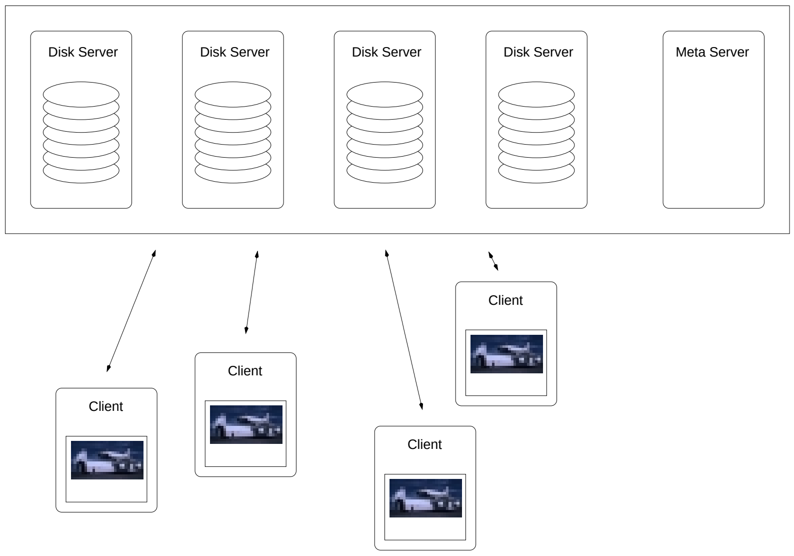


Figure 7.1: Video Server Architecture

7.3.1 Meta Server

The meta server stores and manages all meta information such as the information about the video objects stored, the list of the disk servers storing a video object, the striping and the reliability techniques used, etc.. It also handles all connections between the different video server components, sets up all required connections, and performs admission control for new incoming requests. We have decided for centralized client admission at the meta server in order to ensure deterministic service guarantees, since only the meta server knows about the load and capacity of each of the disk servers and is therefore the only entity that can decide, whether a new incoming client can be admitted.

Besides the functions cited above, the meta server implements the called *striping tool* that is responsible for data striping. The striping tool cuts a video object into multiple video segments (video blocks) [1] and stores these segments sequentially across the different disk servers as illustrated in Figure 7.3. We consider MPEG-1 system streams at the playback rate of $1.5\ \mathrm{Mbit/sec}$. Obviously, determining the segment size implicitly determines the service round duration. For instance, if we set the segment size to $1.5\mathrm{Mbit}$, the service round duration should not exceed the value of $1\mathrm{sec}$. In our case, the segment size is a design parameter within the striping tool.

The striping tool operates as indicated in Figure 7.4. First, the striping tool checks whether there is a new video object to store. Video blocks are generated by splitting the video object. Subsequently, a *video block header* (*vbi*) is created for each video block. *vbi* contains the current video time and the length of the video block, the stream *id* that indicates whether the block is an

[1]As already mentioned, the terms video segment and (video) block are equivalent for the CGS algorithm used.

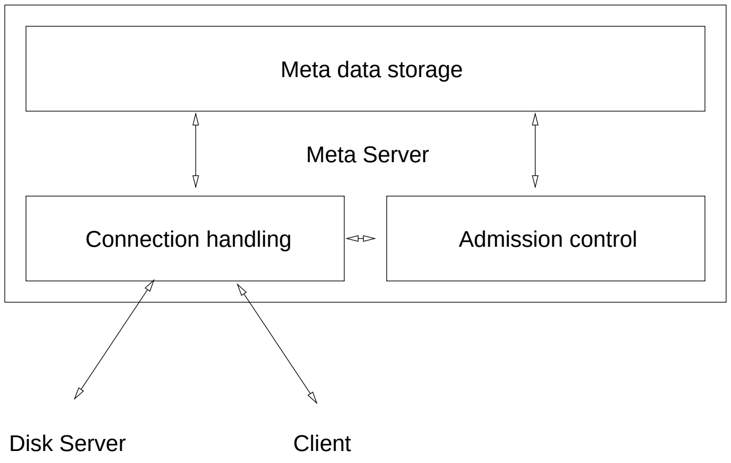


Figure 7.2: Meta Server Tasks

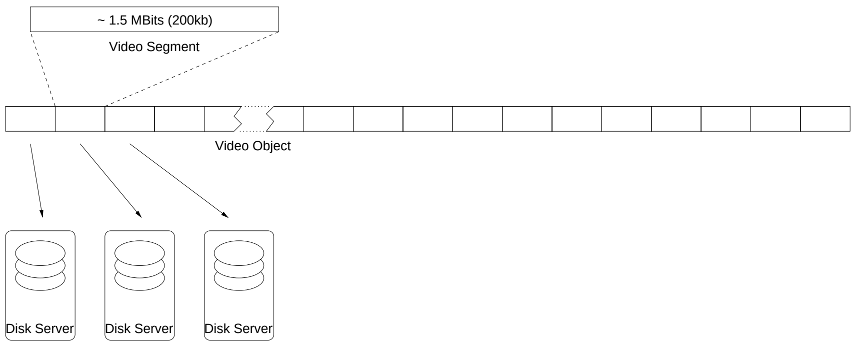


Figure 7.3: Striping Tool at the Meta Server

original or a replicate one [2], the playback duration of the block, the round index that determines the relative time at which the block must be played, and the redundancy information. Once the block header is generated, the video block can be associated to a disk server and a disk. Now, the *vbi* of the replicated block is created, simply by altering the stream *id* value. Thereafter, the replicated block is, in turn, associated to a disk server and a disk that are, obviously, different from those storing the respective original video block.

Figure 7.4 indicates how *original* and *replicated* video data are striped across the video server components. As mentioned before, we have implemented is the CGS algorithm for data striping. For data replication, we have implemented a variety of algorithms including the One-to-All, One-to-Some, and Grouped One-to-One schemes, where different video objects to store may be replicated differently.

[2]The *id* is 0 for original blocks and is 1 for replicated blocks.

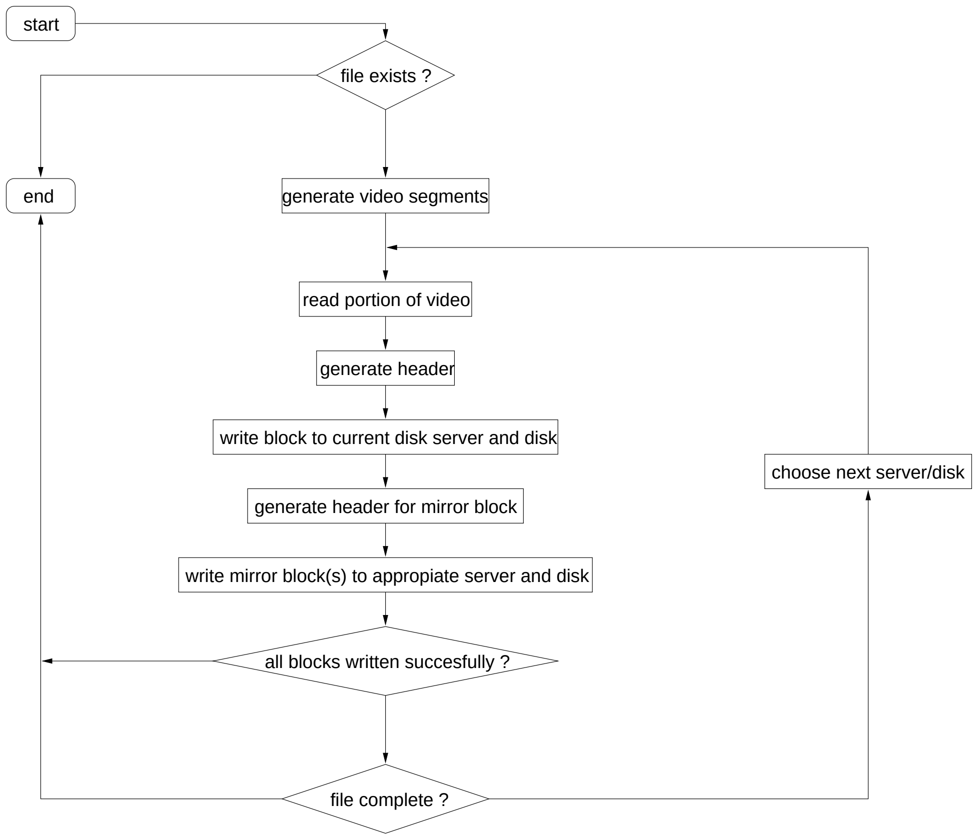

start
file exists ?
end
generate video segments
read portion of video
generate header
write block to current disk server and disk
choose next server/disk
generate header for mirror block
write mirror block(s) to appropiate server and disk
all blocks written succesfully ?
file complete ?

Figure 7.4: Striping Tool Algorithm

7.3.2 Disk Server

The video server is made of multiple disk servers, each of them containing a set of disks. A disk server has two primary tasks. The first task consist in storing and managing portions of several video objects. The second and main task of the disk server is provided by its *scheduler* that reads data from disks and sends them to the clients while guaranteeing continuous video data delivery. Figure 7.5 globally illustrates the main tasks of the disk server. Besides these tasks, the disk server performs disk maintenance, where it executes meta server commands such as writing to disks, renaming or deleting video objects, etc..

Distributed Stream Scheduling

In chapter 2, we have made the choice for a *distributed* video server architecture to ensure high performance and scalability. Besides this distributed hardware configuration, we also implement *distributed scheduling*, where each disk server runs its own stream scheduler without the need to exchange any information with the other disk servers. The stream scheduler at the disk

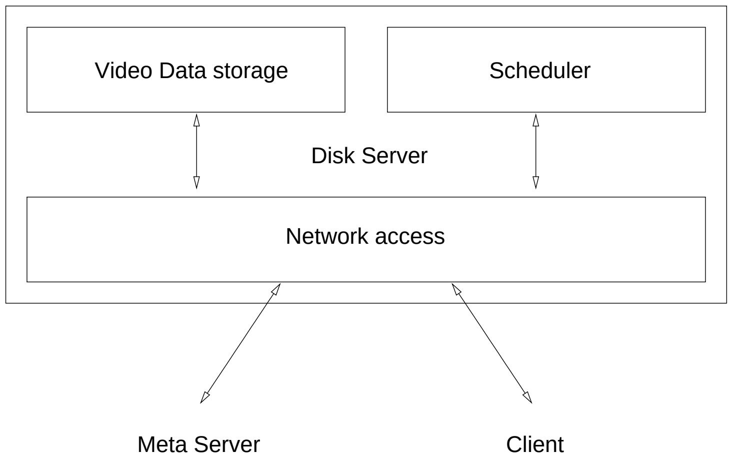


Figure 7.5: Disk Server Tasks

server is the subject of what follows.

Before starting to describe the stream scheduler, it it worthy to note the following: The head of video block *vbi* contains a field, called *timestamp* that indicates the current video time of that block and thus the location of that block in the overall video stream. For each stream, the disk server associates a timer, called *videotime*, that is increased with each service round to reflect the playback time of the corresponding video. It is also important to note that the header of a video block is *not* stored adjacently to that block, but to the one that precedes that block and is stored on the same disk. In other terms, when a block is read from disk, the header (*vbi*) of the next block that has to be retrieved from this disk is simultaneously read. This allows to know in advance, which blocks are to be retrieved in future service rounds.

The scheduling algorithm is illustrated in Figure 7.6. The scheduler starts by building the called *service list* that contains all active *services* (playing or paused). Each service contains the name of the video object consumed and the destination address and port number of the respective client. For each element of the service list, the scheduler checks the respective *vbi* to decide whether a block has to be read from disk. The read operation takes place only if the service is playing and the following holds: $timestamp < videotime$. After having read a block, the scheduler passes it to the network interface that sends it to the client. Additionally, the scheduler reads the retrieved *vbi* that corresponds to the header of the next block to be retrieved from the same disk. This retrieved *vbi* is thereafter inserted into the scheduler service list.

As Figure 7.6 shows, the send operation of a video block can be done *synchronously* or *asynchronously*. Contrarily to the synchronous mode, where read and send operations are performed by the same thread, the asynchronous mode separates the read and send operations and assigns a thread to each of them. The asynchronous mode requires more buffer to put the already retrieved blocks in the send-queue, which is then emptied by the send thread. It enables the disk server

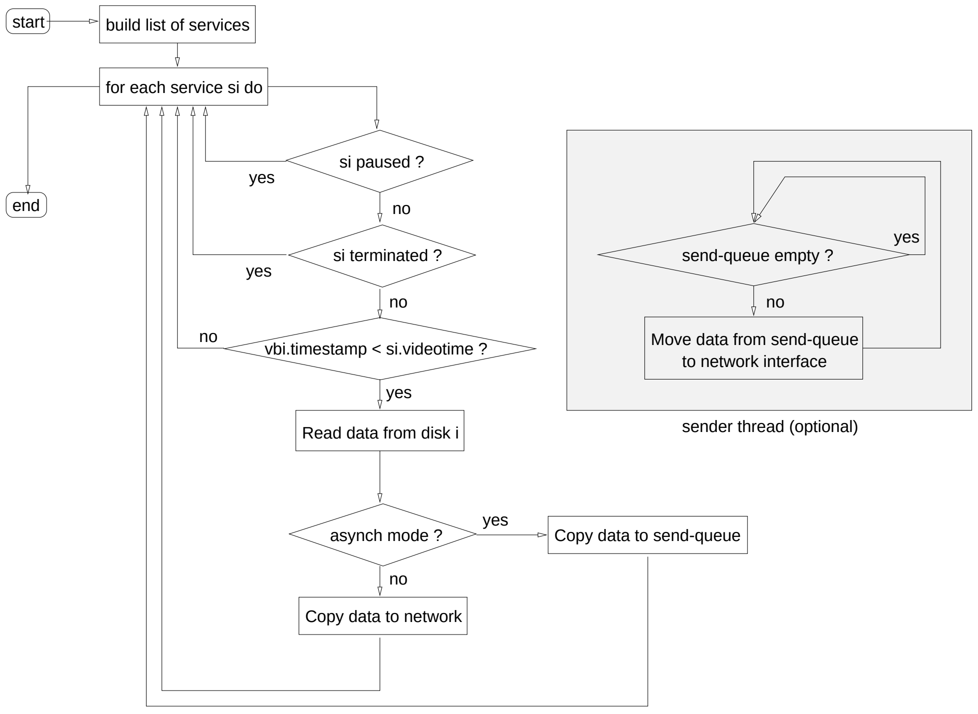


Figure 7.6: Stream Scheduling Algorithm

to separate read and send operations and thus increases parallel processing of video data, which logically results in significant performance increase as compared to the synchronous mode.

7.3.3 Server Client

The client can be considered as the user interface to the video server. It provides the user with an interface to the video server's functionality. Thereby, it shows the list of the available videos as well as the VCR-control interface, receives video data and passes it to the video player process. It also records user interaction and passes it to the video server via its network interface. Figure 7.7 illustrates a schematic overview of the client's tasks.

Platform-Independent Video Server Client

We have implemented a *platform-independent* client using the Java programming language [Mancini 99]. This platform-independence concerns the *client code* itself and also the *MPEG stream decoder*. In fact, there is for the Java code no need to maintain and compile multiple versions of the client code on multiple platforms (e.g. PC running NT, PC running Linux, or workstation running Unix) as is the case for C or C++ codes. For stream decoding and video playback, the client uses the *Java Media Player* (JMP) package of the *Java Media Framework*

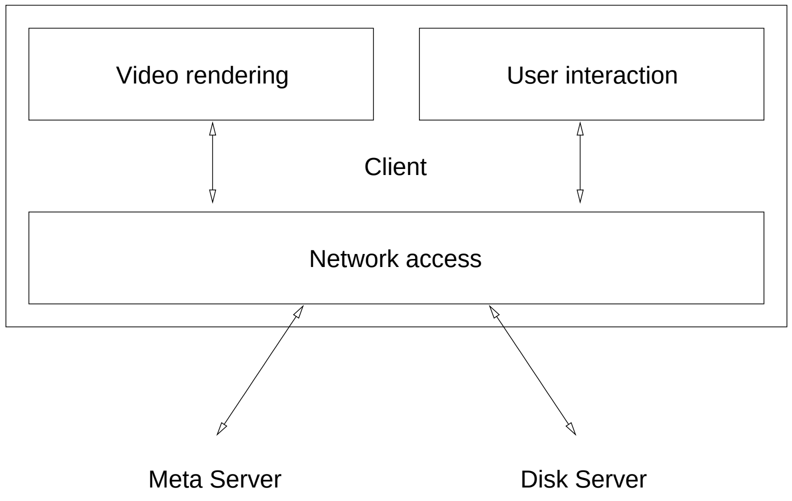


Figure 7.7: Client Tasks.

(JMF) library [3]. The JMP package decodes and visualizes incoming MPEG1 streams independently of the platform, on which the client runs. Note that the MPEG1 decoder is a *software decoder* and JMF allows for full screen on PCs as well as on workstations.

Video playback is organized as follows. Due to the CGS algorithm, the client receives *one* video block during each service round, say during each second. Let us now consider the following situation. The client has been admitted at the meta server and the disk servers are ready to deliver video blocks to that client. Each of the blocks received at the client is immediately put into the buffer. The client is allowed to start with playback, only if the buffer is full. We give later the reason for doing so. The playback is ensured by the JMP that sequentially reads the video blocks from buffer and then visualizes them on the screen after performing the necessary MPEG decoding.

We discuss now, why buffer is needed at the client and how large the buffer must be. Buffer is required at the client to deal with some network delays and heterogeneity that may exist between different disk servers. Indeed, buffer at the client allows for reordering some video blocks that may be delayed. The size of the client buffer is a design parameter and must satisfy the following two *conflicting* constraints. Buffer size must be *high enough* to well eliminate block delays and thus perform reordering within deadlines. On the other side, the duration of the setup phase and thus the delay of playback is a function of the client buffer size: As mentioned, video playback is delayed until the client buffer is full. Hence, the delay equals the maximum number of blocks put in the buffer times the service round duration. Consequently, increasing the buffer size increases the playback delay and therefore the start-up latency at the

[3] The Java Media Framework (JMF) library contains a set of packages for capture and visualization of multimedia data.

client. As a result, the buffer size must be kept *small enough* to get low start-up latency.

The client further provides the user with an interface to the video server functionality. The interface, as Figure 7.8 indicates, shows the list of the available videos, shows and handles VCR-control functions, indicates the total duration and resolution of the current video and its current playback time. Finally. the interface gives the user the option to activate and deactivate the mirroring-based reliability module. This last option is benefit during test and validation phases.

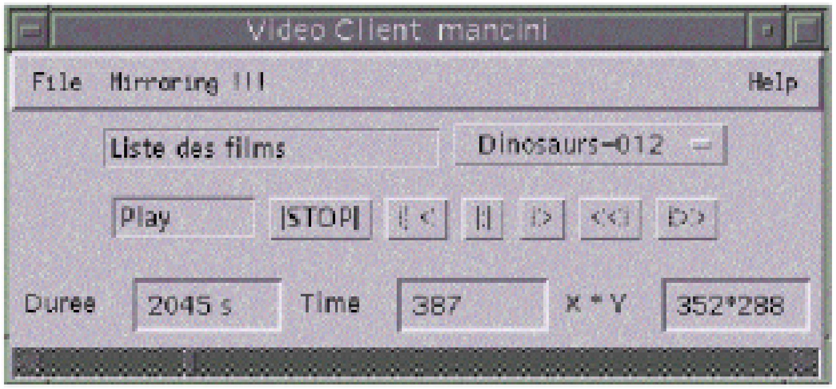


Figure 7.8: User Interface at the Client.

Information Flow

In order to service a client, some information must be exchanged between the client and the video server, which is referred to as *client-server communication*, and between the meta server and the disk servers, which is referred to as *intra-server communication*. Figure 7.9 demonstrates the client-server and the intra-server communications. Further, we distinguish two types of information flow; *video data flow* and *control message flow*. The former is sent by the disk servers to the clients, whereas the latter represents the exchange of the control messages between the clients, the disks servers, and the meta server such as connection setup, video lists, VCR-control messages (e.g. stop, play, pause, etc.), and maintenance commands. *Video control flow* is between the client and the disk servers and is initiated by the client to request a VCR function like reposition, pause, play, etc.. *Meta information flow* is between the meta server and the client and is generated by the meta server during client admission. *Server information flow* is between the meta server and the disk servers to issue commands and exchange meta information. For control message flows, which are much less time-critical than video data flow, TCP connections are used. Instead, in order to satisfy the real-time requirements of video stream delivery, UDP is used for video data flow.

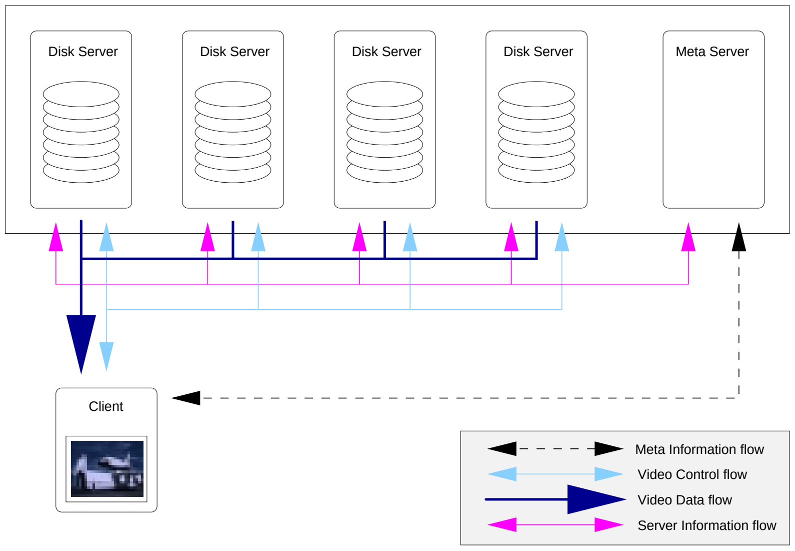


Figure 7.9: Information Flow between the Server Components

7.4 From the Client Request to the Video Playback

After describing the different components of the video server prototype and their functionalities, we look now at how a new incoming client gets connected to the meta server and the disk servers. We then focus on the message flow between client, meta server, and disk servers during the setup phase.

Client Request Management

There are mainly five steps for managing a new incoming client request. Figure 7.10 illustrates these steps:

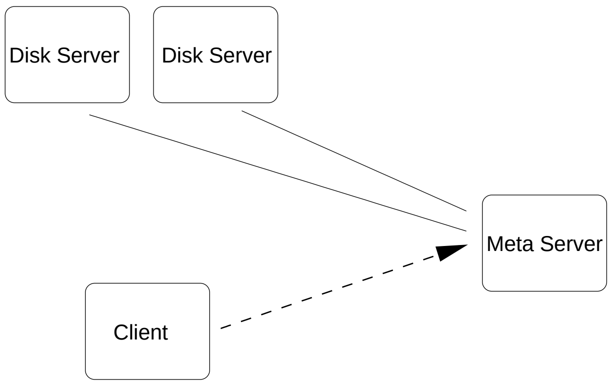
- The client requests a video object (Figure 7.10(a)): The client sends a *MSG_CLIENTOPENSTREAM* message to the meta server. The meta server queries the client for its address and port numbers, later used as connection points for the disk servers. It opens the port and waits for a given timeout period.
- The meta Server identifies the video object (Figure 7.10(b)): The meta server determines the appropriate disk servers and sends them a *MSG_SERVERADDCLIENT* message along with the clients address, port number, and the name of the video object requested.
- The disk Servers connect the client (Figure 7.10(c)): The disk servers try to open connections to the client.

- The client issues a PLAY command (Figure 7.10(d)): The client sends a *MSG_STREAM_PLAY* message to all connected disk servers via the newly established low-bandwidth control connections.
- The disk Servers send the expected video data (Figure 7.10(e)): Via the established high-bandwidth connections, the disk servers transmit the video blocks to the client, where they are then displayed.

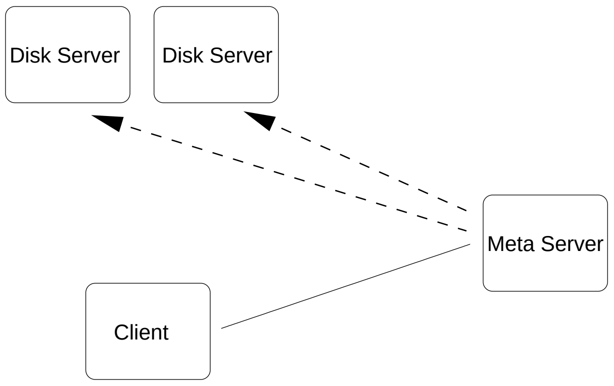
Message Flow during Client Setup

At the start up of the video server, the disk servers establish connections to the meta server (intra-server communication) and thus enabling message exchange within the video server. A starting client connects to the meta server that responds by sending to that client the list of the available video objects. Until now, the disk servers are connected to the meta server and the client is, in turn, only connected to the meta server. All of these connections are referred to as *initial connections* and are responsible for carrying all the message flow between the three components (meta server, disk servers, and client), but not for video data. Thus, they can and should be established over relatively low-bandwidth networks using TCP as transport protocol. Let us take the message flow example illustrated in Figure 7.11. Thereby, the client on the left wants to open a video stream. This is carried out as shown below:

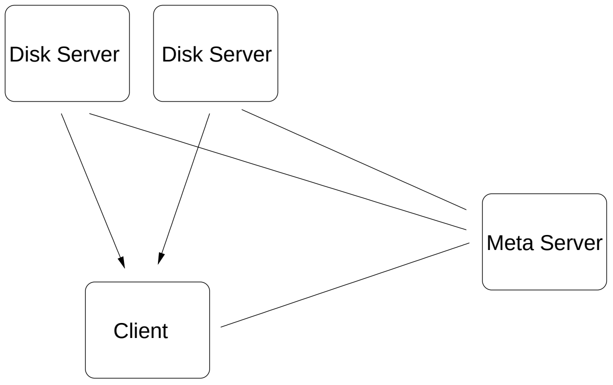
- The user's request for opening a video stream results in the execution of a client procedure doing the following:
  - Pass `MSG_CLIENTOPENVIDEOSTREAM` to the meta server
  - Pass the name of the video, the host name and port number
- The meta server determines the appropriate disk servers that store the requested video object. It then instructs them to open connections to the client and to add the client to their service list by doing the following:
  - Pass `MSG_SERVERADDCLIENT` to the respective disk servers
  - Pass host name and port number of the new incoming client and the name of the requested video.
- Each disk server now tries to establish two connections with the client by connecting to the port supplied in the message. On success, the disk server passes its handle over to the meta server and if not it passes $FALSE$.



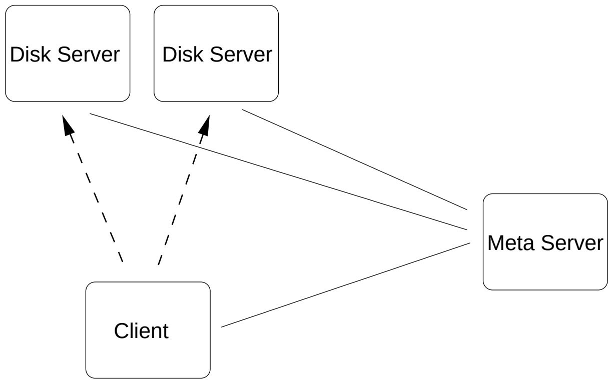
(a) Client requests video object.



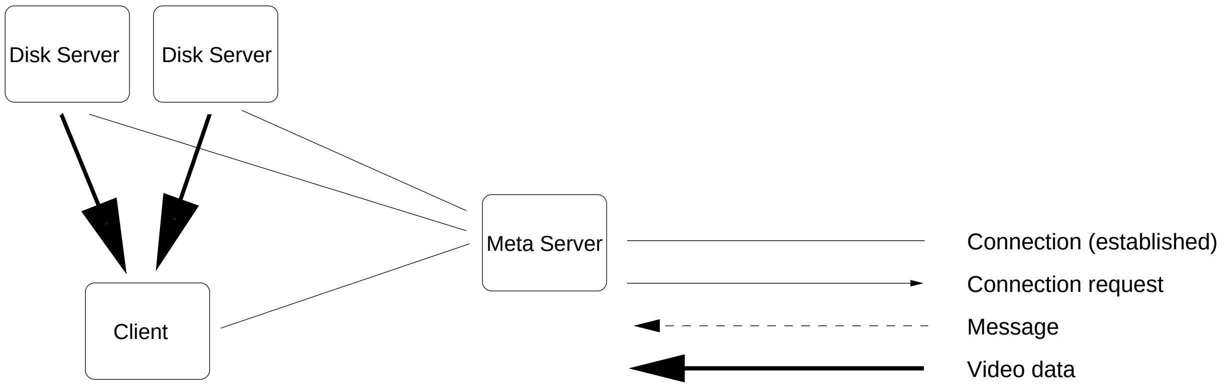
(b) Meta Server requests video object.



(c) Disk Servers connect.



(d) Client issues PLAY command.



(e) Disk Servers send video data.

Figure 7.10: From Client Request to Video Play-back.

- The client accepts all connections while waiting in a loop (for a given timeout such that it will not lock up).
- The meta server collects the responses and passes the handle over to the client on success, or $FALSE$ on failure.
- The client's procedure returns control to the main loop, passing the result.

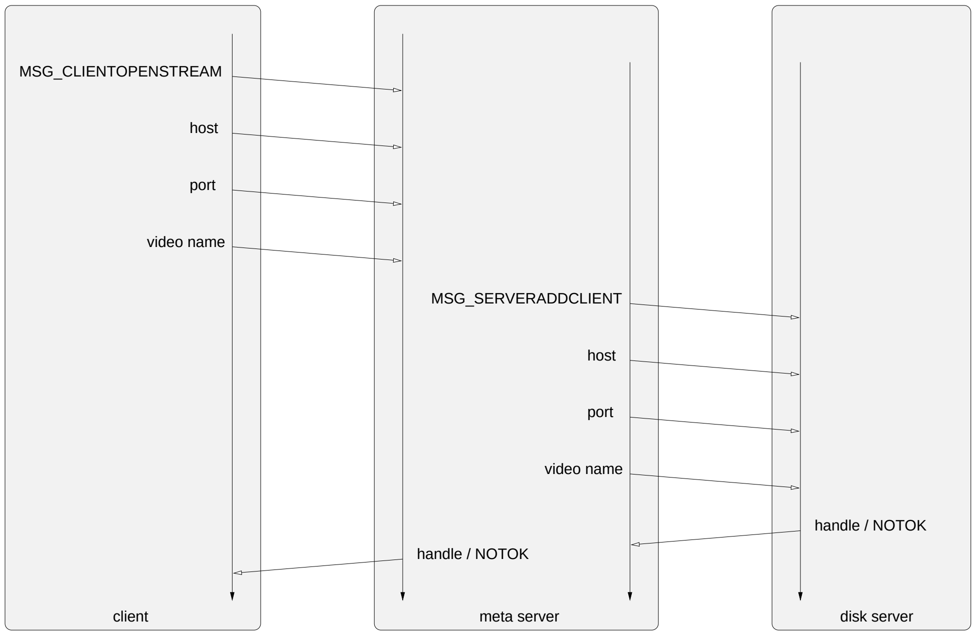


Figure 7.11: Setup Message Flow Example

7.5 Operating in Failure Mode

The video server prototype implements mirroring-based reliability. Only the meta server has the knowledge about where the replication of a given block is stored on. Further, the detection of disk or even node failures is performed at the client side. Once a failure is detected, the client informs the meta server about the blocks that have been lost. The meta server is then able to determine, which disk has failed. Having this information, the meta server instructs the affected disk servers [4] to open a mirror stream for that client. We explain as follows how the detection of disk/node failure is performed at the client and the behavior of the meta server, the disk servers, and the client when operating in failure mode.

[4]The affected disk servers are those containing replica of original blocks that reside on the failed disk.

7.5.1 Client-Based Failure Detection

Figure 7.12 shows the algorithm we use to detect block loss and disk failure. It includes the following steps:

- The client opens a video stream by sending a request to the meta server.
- The client receives the video info, which contains the number of disks D, on which that video is striped.
- The client builds an array $lossdetect[]$ of size D and initializes each field with the maximum number $n_{maxlost}$ of blocks that are allowed to be lost consecutively before informing the meta server about disk failure.
- The block ordinal number counter $expected$ is set to 0.
- The client is receiving video blocks.
- By looking at the block headers, the client gets the blocks timestamp and its ordinal number $sequence$.
- If the ordinal number is higher than the expected number ($sequence > expected$), there is a block missing.
- Video data is striped round-robin, so a simple modulo-operation gives us the (virtual) number of the disk and thus the position pos within the $lossdetect$-array:
  $pos = expected \bmod D$
- $lossdetect[pos]$ is decreased by 1 to reflect the missing block.
- $lossdetect[pos]$ is checked. If it is still in range (> 0) we just go on with the next block.
- If $lossdetect[pos] <= 0$, the client assumes a disk failure and informs the meta server as we will see in section 7.5.2

Based on the algorithm presented in Figure 7.12, the failure detection depends on the values of D and $n_{maxlost}$. The client experiences interruptions due to the loss of $n_{maxlost}$ blocks and it will take $D \cdot n_{maxlost}$ blocks until it can detect the failure. Thus, the value of $n_{maxlost}$ is chosen as small as possible, but high enough to be sure that the losses are not due to network delays. We took the value 3 for $n_{maxlost}$.

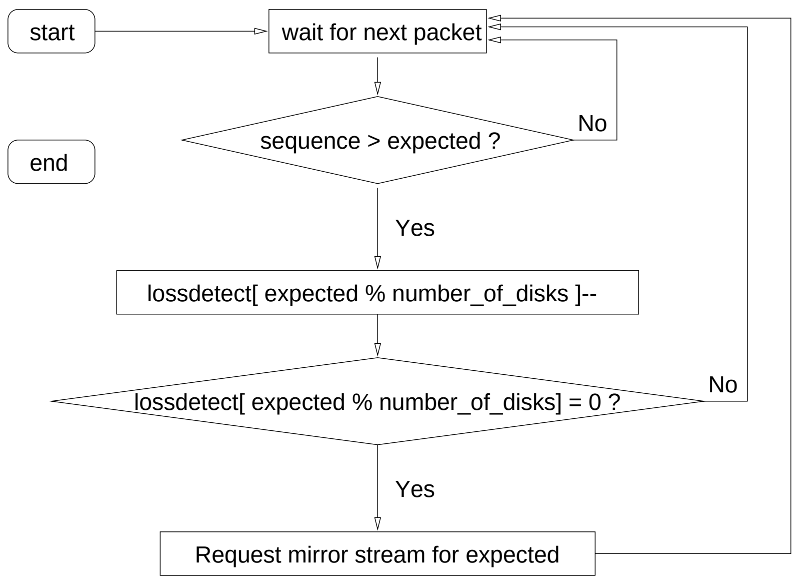

start
wait for next packet
end
sequence > expected ?
No
Yes
lossdetect[expected % number_of_disks]--
lossdetect[expected % number_of_disks] = 0 ?
No
Yes
Request mirror stream for expected

Figure 7.12: Client-Based Failure Detection

7.5.2 Retrieval of Replicated Blocks

We describe in the following the behavior of the client, meta server, and disk servers when a disk failure occurs.

Client

The client attributes to each disk a counter that is initially set to 0. When the client detects a lost block, it increments the corresponding counter, which denotes the disk that contains that lost block. When one of the counters attains the value $n_{maxlost}$, the client sends a request to open a mirror stream to reconstruct the failed part of the original stream by issuing a `MSG_CLIENTOPENMIRRORSTREAM` command. The sequence of messages passed leads to the following:

- The client issues an `MSG_CLIENTOPENMIRRORSTREAM` command and passes the ordinary number $expected$ of the last missing block as a parameter.
- The meta server calculates a valid disk server hosting a mirror stream.
- The meta server issues a `MSG_SERVERADDMIRROR` command to the appropriate disk server.
- The disk server adds the stream to its service list.

- The meta server repositions all streams to position $expected$.
- The client receives the expected blocks.

Due to the *reactive* nature of the failure detection algorithm we use, the client will encounter an interruption. Two alternatives are possible for repositioning and restarting playback after failure correction: the first alternative repositions the stream to the first lost block, whereas the second alternative, which we have implemented, repositions the stream to the last block lost. Note that in the case where the meta server is not able to open a mirror stream, it returns $FALSE$ as the result of the operation. The client will *not* stop playback but continues viewing the video stream that contains periodical interruptions.

Meta Server

The meta server, on reception of a `MSG_CLIENTOPENMIRRORSTREAM`, queries the originator for its name and port number as well as the video stream name and the ordinary number $sequence$, at which loss was detected, and the stream id. The id field is always 0 for original streams and > 0 for mirror streams, so it can be used at the meta server to check whether a mirror stream is available for a given original stream. Figure 7.13 shows the meta server algorithm.

- The meta server receives a `MSG_CLIENTOPENMIRRORSTREAM` from the client.
- The video stream name, $sequence$, id, and the client's host name and port number are passed.
- If the id is higher than the available number of mirror streams held in $vi.groupinfo.redundancy$, the meta server returns $FALSE$ as result of the operation.
- Knowing the video is striped in round-robin fashion and the ordinary number of the block $sequence$, the meta server can calculate the original server and disk.
- The meta server is now able to determine the disk servers and disks that contain replicated blocks of original blocks that reside on the failed disk.

  The same algorithm used during striping of videos returns the server and disk where copy id of the stream is stored.
- If the disk server is active, the meta server issues a `MSG_SERVERADDMIRROR` and passes the client's host name and port number with the name of the mirror stream and the disk over to the disk server.

- If the disk server is not active or the former step is not executed correctly (i.e. no response from disk server), the meta server increases the id and tries again.

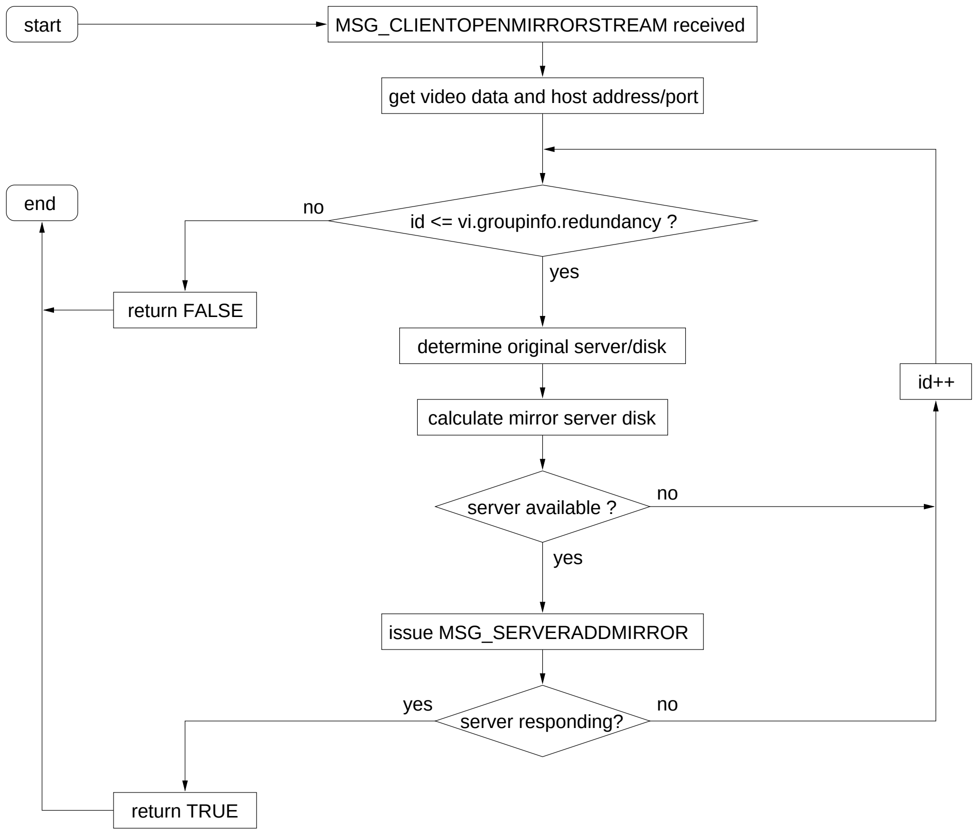

start
MSG_CLIENTOPENMIRRORSTREAM received
get video data and host address/port
end
no
id <= vi.groupinfo.redundancy ?
yes
return FALSE
determine original server/disk
id++
calculate mirror server disk
server available ?
no
yes
issue MSG_SERVERADDMIRROR
yes
server responding?
no
return TRUE

Figure 7.13: Opening a Mirror Stream (Meta Server)

Disk Server

After (i) the client has detected losses of original blocks and requested a mirror stream and (ii) the meta server has determined the appropriate disk servers that must reconstruct lost data, each of these disk servers receive a `MSG_SERVERADDMIRROR` with the client's host name and port number, the mirror stream name, and the disk index. Remind that the video server implements the three mirroring schemes One-to-All, Grouped One-to-One, and One-to-Some. For all of these schemes, a disk server containing replicated blocks of a video object, obviously contains original blocks of that video object. As a result, the disk server opens a mirror stream for a client only if that client is already contained in the disk server's service list. To deliver the mirror stream, the disk server must add a new entry to the service list. The algorithm implemented for retrieving replicated blocks from the disk server is shown in Figure 7.14.

- The disk server receives a `MSG_SERVERADDMIRROR` from the meta server.
- The video stream name, $sequence$, id and the client's host name and the port number are passed.
- The disk server looks up the already existing service representing the client.
- The disk server checks for availability of the mirror stream.
- The mirror stream is added to the existing service.
- The disk server returns $TRUE$ to the meta server.

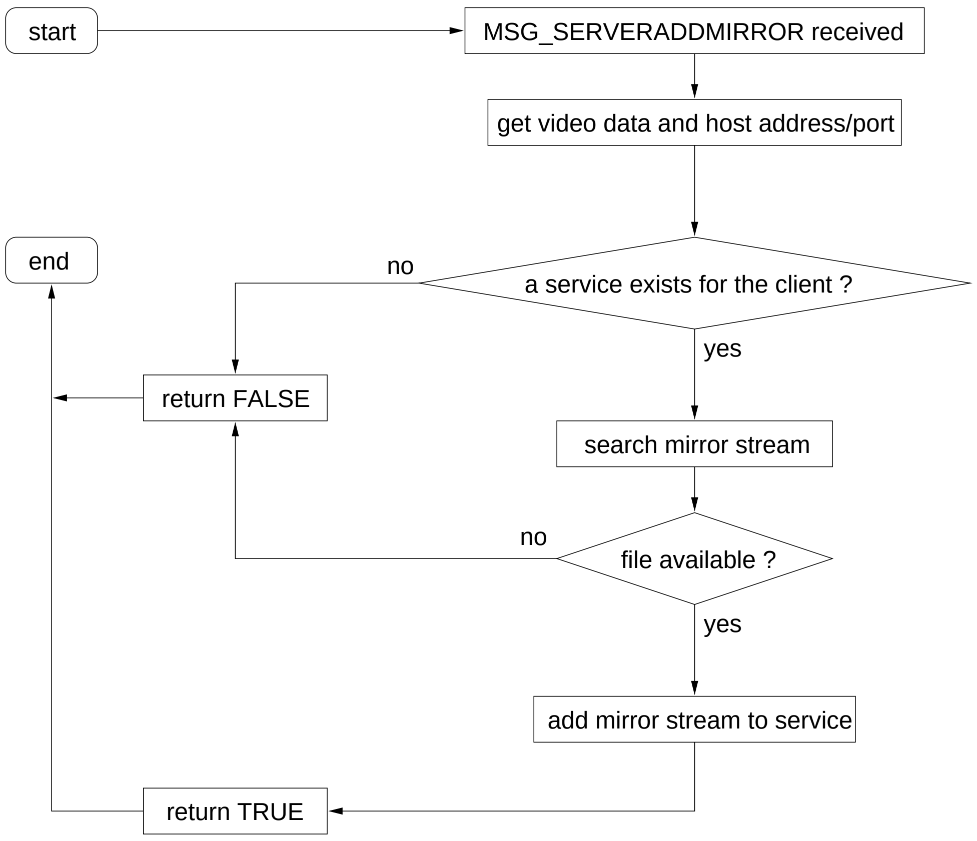

start
MSG_SERVERADDMIRROR received
get video data and host address/port
end
no
a service exists for the client ?
yes
return FALSE
search mirror stream
no
file available ?
yes
add mirror stream to service
return TRUE

Figure 7.14: Opening a Mirror Stream (Disk Server)

7.6 Performance of the Video Server Prototype

This section evaluates the performance of the video server prototype we have implemented. From the design point of view, we have adopted in the previous chapters the solutions that ensure video server *scalability* such as the *server array architecture* and the *video wide striping technique*, e.g. the CGS algorithm. These video server design solutions only indicate that the video server is scalable with respect to the *hardware*, which means that video server performance in terms of server throughput increases as new components are added to the server. Theoretically, the performance increase is linear to the hardware growth. However, the fact that standard hardware is used to build a cost effective video server shifts the whole video server complexity to the application level and thus to the *software*. As a result, both, the hardware as well as the software are candidates to be the potential bottleneck of the video server. Hence the need for our implementation to determine (i) the exact performance of the server in terms of server throughput for a given hardware capacity (number of disks and their I/O bandwidth capacity, number of disk servers) and (ii) the server scalability by considering the evolution of the server throughput when new components are added to the video server. This section investigates video server throughput and scalability. The rest of this section is organized as follows. Section 7.6.1 gives a background of the hardware used and its possible bottleneck. Finally, section 7.6.2 hands out the video server performance results.

7.6.1 Possible Bottlenecks

In order to avoid a software bottleneck due to a high processing overhead at the meta server, the functionality of the latter has been restricted. Indeed, the meta server functionality is limited to the client connection setup and to open-mirror stream commands during failure mode. However, the disk server can present either a hardware bottleneck or a software bottleneck. We first examine the hardware characteristics of the disk server. The characteristics of a Sparc20 workstation are illustrated in Figure 7.15. Given that the network side is not considered, the Figure indicated that the I/O bandwidth of the SCSI interface has the smallest value . As the weakest part of the machine determines the maximum speed of that machine, the I/O bandwidth of the SCSI interface presents the machine bottleneck (hardware bottleneck). Consequently, the maximum number of disks that are connected to the same machine (disk server) is limited by the capacity of the I/O bandwidth of the SCSI interface. More precisely, the sum over the data transfer rates of all disks that are connected to the same machine should not exceed the I/O bandwidth of the SCSI interface.

7.6.2 Video Server Performance: Results

Our performance study neglects network aspects and only concentrates on the throughput that the video server (disk servers and meta server) is able to achieve. For video data storage, we use two different SCSI disk families. The first disk family is the Micropolis $4110AV$ that has $1\ \ GByte$ storage capacity and a transfer rate of up-to $40\ \ Mbit/sec$. The second disk fam-

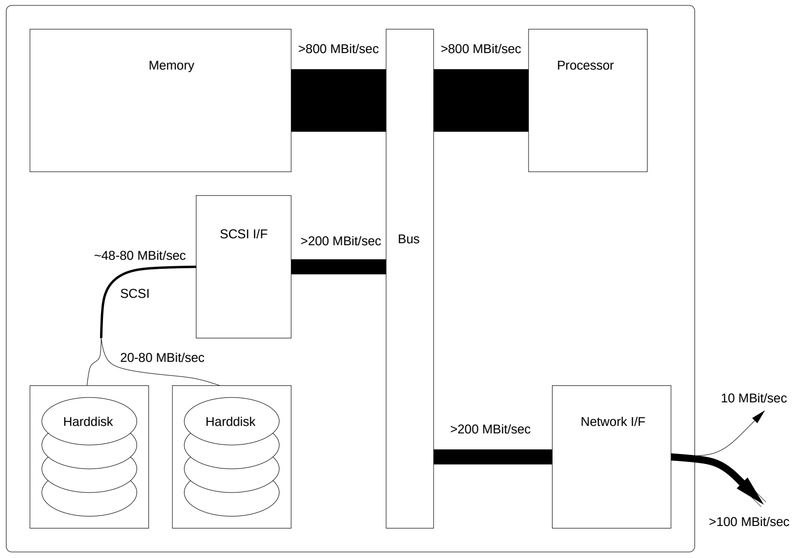


Figure 7.15: Hardware configuration of a disk server.

ily is the Fujitsu $2952S - 512$ that has $2\ \mathrm{GByte}$ storage capacity and a transfer rate of up-to $80\ \mathrm{Mbit/sec}$. For our experiments, we have four slow (Micropolis) disks and four fast (Fujitsu) disks. We use various video server configurations that involve Sparc10, Sparc20, Ultra1 workstations running Solaris, and pentium PCs running NT. Our performance goals are (i) to determine the *throughput* that a given server configuration can achieve and (ii) to demonstrate whether the video server prototype is *scalable*. We use the following scenario. We connect each of the 8 disks to a single machine (workstation or PC) and thus build 8 server nodes. We add to the video server progressively server nodes and measure the throughput achieved with each configuration. The results are illustrated in Figure 7.16. For a video server made of *one* server node, the throughput is 12 clients for a slow disk and is 22 clients for a fast disk. Remind that each client is consuming MPEG1 coded streams at a rate of $1.5\ \mathrm{Mbit/sec}$ The two curves in Figure 7.16 present two different scenari. By the first one, we have added first the slow disks and then the fastest disks within the server. We did the opposite by the second scenario. The Figure shows that the video server prototype is scalable in terms of the number of server nodes. It also shows that the server throughput linearly increases for a homogeneous configuration, e.g until 4 slow nodes or until 4 fast nodes. However, as far as the video server contains hybrid nodes, the increase in server throughput becomes non linear. This mainly due to the round-based data retrieval that requires from all nodes to retrieve data within the same deadlines and therefore, the slow nodes are the server's bottleneck and the fast ones experience high idle times.

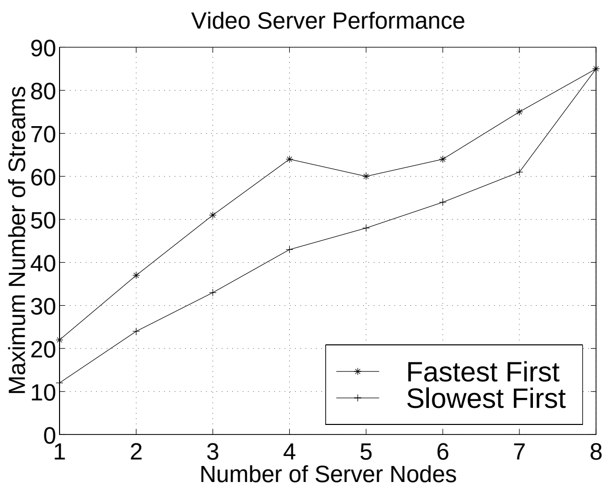


Figure 7.16: Video server performance.

7.7 Summary

In this chapter, we have described the main characteristics of the video server prototype we have implemented at Eurecom (see also [Walter 97, GAFS 98b, Exner 99]). The video server prototype is made of standard (and hence cheap) hardware and is *multi-platform*, where the server nodes run on workstations running Solaris as well as on PCs running NT or Windows 95. The client is implemented in Java and is therefore platform-independent. The prototype performs distributed stream scheduling and data delivery and ensures fault-tolerance with data replication. Both, disk and node failures can be tolerated. Furthermore, failure detection and data recovery are implemented in software and are managed at the application level. Moreover, We have evaluated the server performance, where the results have shown that the video server is scalable in terms of the number of server nodes. However, the increase of the number of server nodes does not lead to a linear increase in server throughput. We are currently optimizing the different components and algorithms of our prototype in order to achieve a linear performance behavior and therefore total server scalability. Eurecom's video server prototype should serve as a test application to validate France Telecom's ADSL experimental platform.

Chapter 8

Conclusions

Rapid advances in computing and communication technologies led to an explosive growth of multimedia services. A variety of these services will become increasingly popular in the near future such as multimedia messaging, interactive television, and video-on-demand. The realization of such services requires the development of large scale systems, i.e. video servers, that are capable of storing, retrieving, and transmitting data to thousands of users. In this dissertation, we proposed, compared, and validated several algorithms and schemes that are involved in the different design aspects of a distributed and reliable video server. The result is a complete video server design that addresses all requirements of digital video storage, retrieval, and delivery in order to find the best trade-off between video server performance, cost, and reliability.

Throughout this thesis, we have learned that the different aspects that are involved in the design of a video server are *interdependent* and must be addressed together. One example of this statement is the relationship between the data striping algorithm and the server reliability technique used, which was emphasized in this thesis. Furthermore, this dissertation showed that mirroring is a very attractive technique to ensure fault-tolerance for video servers. Indeed, one -at first glance- surprising result indicated that mirroring is more cost effective than parity. Finally, the work in reliability modeling has not only allowed to quantitatively evaluate and compare several reliability schemes for video servers, but it also guided us to propose the Grouped One-to-One scheme that achieves *highest* video server reliability. Last but not least, the video server performance, cost, and reliability analysis we performed gave us the intuition to propose the replica placement scheme ARPS that improves the video server performance by up-to 90% as compared to existing replica placement schemes. In what follows, we first summarize the contributions of this dissertation and then explore some open issues.

In this dissertation, we made *five* primary contributions. Below, we describe our contributions in detail.

We studied *data striping* for non-fault tolerant video servers. We proposed a striping algorithm, called *MGS (Mean Grained Striping)* that consists in dividing the video server into independent groups, where disks belonging to the same group are contained on different server nodes. We compared MGS with the two well known striping algorithms FGS (Fine Grained Striping) and CGS (Coarse Grained Striping) in terms of server throughput, buffer requirement, and start-up latency. The results showed that FGS suffers under buffer explosion and thus achieves a much lower throughput than CGS and MGS for a given buffer space. CGS has highest server throughput for the same amount of buffer, whereas MGS has lowest start-up latency. We then introduced video server fault-tolerance and compared CGS and MGS when considering both, parity-based reliability and mirroring-based reliability. We analyzed the video server performance during disk failure and discussed reactive and preventive modes for reconstruction of lost data. We found out that *data striping and server reliability are related*: Our results indicated that CGS combined with mirroring-based reliability and MGS combined with parity-based reliability are the best combinations with respect to the overall server throughput.

We looked at performance and cost issues for CGS-striped video server and for different reliability techniques. We classified several video server reliability schemes based on the technique used (mirroring vs. parity) and on the distribution granularity of redundant data. The cost function includes hard disk and main memory costs. we found out that *mirroring-based reliability is more attractive than parity-based reliability*. Indeed, the results indicated that mirroring-based reliability, despite its 100% storage overhead requirement, is more cost effective than parity-based reliability, given that hard disk prices decrease faster than memory prices. Another advantage of mirroring is the disruption time after a disk failure, where mirroring takes at most one service round to send the expected replicated block. Parity, however, takes many service rounds to retrieve all blocks belonging to the parity group of the lost block and thus the disruption time can be very high, which also affects the start-up latency for new incoming clients.

We modeled and analyzed the performance of reliability strategies for distributed and fault-tolerant video servers. We proposed a new mirroring-based scheme, namely the *Grouped One-to-One* scheme that is based on the traditional chained declustering scheme and further divides the video server into independent groups similarly to the orthogonal RAID principle. We distinguished seven reliability schemes that differ in whether parity or mirroring is used and in the distribution granularity of parity/replicated data. We then quantitatively evaluated video server reliability for these retained schemes by performing reliability modeling based on Continuous Time Markov Chains (CTMC). We considered two cases of independent disk failures and dependent components failures. We took various values of component's (disk and node) failure rate and repair rate and found out that *the Grouped One-to-One always achieves highest server reliability* as compared to all other schemes considered. We also realized that schemes that only tolerate a single disk failure, which we called the One-to-All schemes, suffer from very poor server reliability that does not increase even with decreasing values of the component's failure rate and increasing values of the component's repair rate. Based on these results and given that the trend of the coming years predicts larger number of disks, there is a pervasive need for going beyond the reliability provided by these One-to-All schemes. Our focus then concentrated on schemes that tolerate multiple disk failures and even complete node failures as is the case for the One-to-Some parity/mirroring schemes that divide the video server into several independent groups. For these schemes, we studied both, video server performance and reliability. The results showed that for these One-to-Some parity/mirroring schemes, *the smaller the parity/mirroring group size, the better the trade-off between high server reliability and low per stream cost*.

We proposed a novel replica placement scheme for mirroring-based video servers, the called *ARPS (Adjacent Replica Placement Scheme)*. In contrast to the classical replica placement schemes, where each single disk of the server is dedicated to original video data and redundancy data, ARPS splices replicated data and original data and thus does not require any additional seek time and rotational latency when operating in the presence of disk failures. The results showed that ARPS outperforms, in terms of the server throughput, classical interleaved declustering schemes like the one proposed for the Microsoft Tiger video server. In fact, ARPS achieves 60% to 90% more server throughput than these schemes considered.

We instantiated our design decisions in a prototype implementation. The video server prototype implemented addresses the main goals that were followed during the server design phases, namely high server performance and scalability, cost effectiveness, and reliable and uninterrupted service. The prototype is made of various server nodes, each behaving as single disk array. The prototype further runs on *multiple platforms*, where the server nodes run on both, UNIX machines (Solaris) and PCs (Windows NT, Windows 95). To ensure scalability, the video server prototype implements distributed scheduling and data delivery such that server nodes deliver expected video data without the need to exchange any information. To be fault-tolerant, the prototype implements various mirroring-based schemes and disk or node failure detection and data recovery are performed at the application level (software solution). Moreover, component failure detection is shifted to the client, which reduces the processing overhead at the server side and ensures server scalability. Furthermore, the client is implemented in Java and is thus *platform independent* with respect to the code itself and the MPEG1 decoder (the same code is executed and the same MPEG1 decoder software is used on different platforms). The experimental results showed that the video server prototype is scalable in terms of the number of server nodes that are contained on the video server.

Appendix A

Disk Parameters

Table A.1 shows the disk parameters and their values considered. These values are those of Seagate and HP for the SCSI II disk driv es [GKSZ 96].

Parameter	Meaning of Parameter	Value
r_p	Video playback rate	$1.5 \ \mathrm{Mbps}$
r_d	Inner track transfer rate	$40 \ \mathrm{Mbps}$
t_{stl}	Settle time	$1.5 \ \mathrm{ms}$
t_{seek}	Seek time	$10.39 \ \mathrm{ms}$
t_{rot}	Worst case rotational latency	$9.33 \ \mathrm{ms}$
b_{dr}	Block size	different values considered
τ	Service round duration	$\frac{b_{dr}}{r_p} \ \mathrm{sec}$

Table A.1: Performance Parameters

Appendix B

Data Layout and Scheduling for a CGS-striped Server

We present in this section data layout examples for CGS-striped video server when using parity-based reliability (equivalent to RAID5) [1] and mirroring-based reliability. We further present for each of these reliability techniques generic scheduling and retrieval algorithms for normal operation mode and failure operation mode.

B.1 Parity-Based Reliability

Figure B.1 shows for a video server with D disks, how one video object is stored using the parity-based scheme (similar to RAID5): A video object is assumed to be partitioned exactly into N_{vs} video segments where $N_{vs} = Z \cdot D \cdot (D-1), \quad Z \in \{1, 2, ...\}$. Note that for the CGS algorithm used, the video segment size equals the disk retrieval block size and therefore the two notations, video segment and disk retrieval block, are equivalent.

The parity disk retrieval blocks are distributed in a *round robin* fashion. Each original disk retrieval block of a stored video object is identified by its unique number. Let b_s denote an **original retrieval block number**, then $b_s \in [1, ..., N_{vs}]$. Parity blocks are labeled using the *parity group number*. Since each parity group contains exactly one parity block, the **parity block number** equals the parity group number, to which it belongs. For example, the parity disk retrieval block of the original disk retrieval blocks having numbers between 1 and $(D-1)$ (first parity group) has the parity number P_1. Parity blocks have numbers varying between P_1 and $P_{Z \cdot D}$. Note that the numbers of the *original* disk retrieval blocks are relative to *one* video object and are logical numbers. To simplify the discussion, the retrieval of the *first* disk retrieval

[1] We use the terms parity and RAID5 interchangeably in this section.

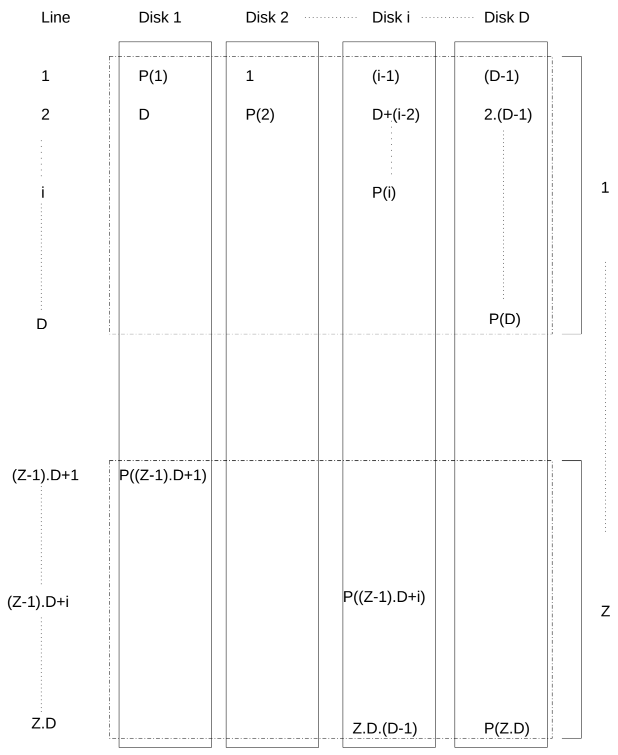


Figure B.1: The RAID5 data layout for one video object stored on a video server with D disks.

block of the video object is assumed to occur during service round 1. In this way, we assign to each client its relative service round, and the service round number equals the current disk retrieval block that must be read from the server. As Figure B.1 shows, the data placement within the server is represented by placing the numbers inside a matrix that contains D columns (*the disks*) and $(Z \cdot D)$ **retrieval lines** (*the parity groups*). The later discussion will be based on this (D, $(Z \cdot D)$)-matrix.

In Figures B.2 and B.3 we show the storage layout of the original and the parity disk retrieval blocks. Figure B.2 gives the number of original and parity disk retrieval blocks that are stored on a given disk i $(i \in [1, ..., D])$ of the server. Figure B.3 gives the storage organization of the whole (D, $(Z \cdot D)$)-matrix, where the diagonals contain parity disk retrieval blocks and the rest original disk retrieval blocks. The scheduling procedures described later use figure B.3 to locate disk retrieval blocks within the matrix (on the right or the left side of the diagonal).

B.1.1 Scheduling and Retrieval for Normal Operation Mode

During normal operation mode, the scheduler retrieves consecutive *original* disk retrieval blocks from consecutive disks. Note that consecutive *original* disk retrieval blocks can be stored on disks i and $i+1$ or on disks i and $i+2$ (the second case holds when a parity disk

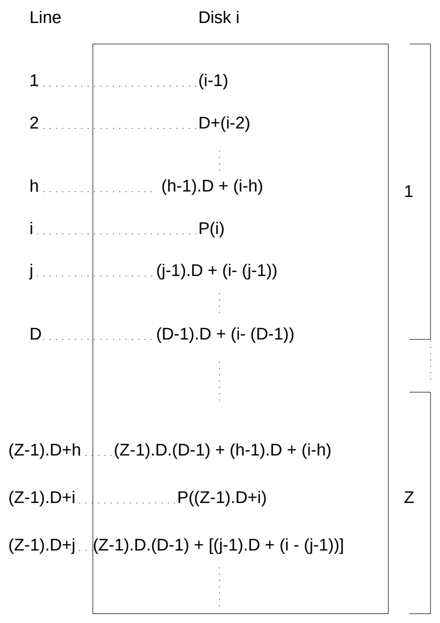


Figure B.2: Storage layout of original and parity data of one video object on disk i.

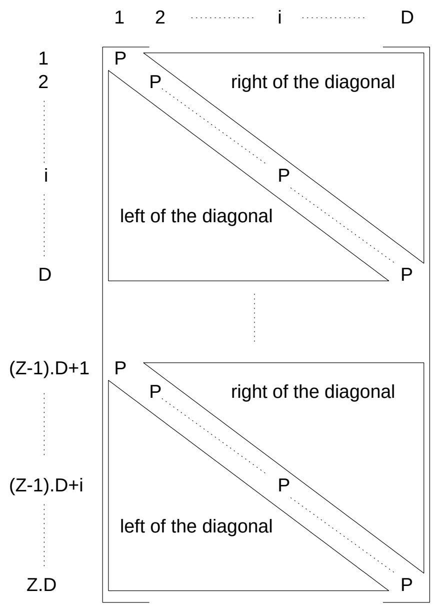


Figure B.3: The (D, $(Z \cdot D)$)-matrix organization.

retrieval block is stored on disk $i+1$ between two consecutive original disk retrieval blocks). Thus, the scheduler needs to know the exact position of each *original* and *parity* disk retrieval block in order to coordinate the retrieval of only original blocks during the normal operation mode. We formulate the problem as follows: the scheduler currently retrieves blocks of the parity group with the number g and has already retrieved an original disk retrieval block from disk $(d-1)$. It must decide, *whether it next retrieves an original disk retrieval block from disk d or not*. Based on the layout order shown in Figures B.1, B.2, and B.3, the scheduler uses `procedure R_seq_no(`d`, `g`, `b`)` to decide. Depending on the parity group number, the disk retrieval block contained on disk d can be a parity disk retrieval block (parity group number equals $(j \cdot D + d)$) that *should not* be retrieved, or an original disk retrieval block that must be read. In the second case, the sequence number of the original block depends on the parity group number. To read a disk retrieval block with the sequence number b from disk d, the scheduler calls the `procedure R_ret_block(`d`, `b`)`.

```
procedure R_seq_no($d$, $g$, $b$)
/* Input parameters */
    /* $d$:  disk number */
    /* $g$:  parity group number */

/* Output parameter */
    /* $b$:  disk retrieval block number (original or parity) */

BEGIN
    $k = g$ mod $D$; /* line of last (D,D)-matrix */
    $z = g$ div $D$; /* number of complete (D,D)-matrices */

    IF (($k == d$) OR ( ($k == 0$) AND ($d == D$))) {
       /* block is parity data block on the diagonal*/
       $b = P_g$;}
       /* NO RETRIEVAL */

    ELSE /* disk retrieval block is an original block */
       IF ($k == 0$) {
          $k = D$; }
       $b = (k-1) \cdot (D-1) + d + z \cdot D \cdot (D-1)$;
       IF ($1 \leq k \leq (d-1)$) { /* right of diagonal */
          $b = b - 1$; }
       ret_block($d$,$b$); /* read block $b$ */
END
```

B.1.2 Scheduling and Retrieval for Single Disk Failure Operation Mode

During a single disk failure, the retrieval blocks stored on the failed disk are **lost** and must be reconstructed using the remaining $(D-2)$ original blocks and the parity disk retrieval block that belongs to that parity group. Such a parallel read requires the temporary storage of $(D-1)$ retrieval blocks for one client. In order to keep the time during which the $(D-1)$ blocks are stored in memory as short as possible, our RAID 5 scheme performs a *second read* of the *original* blocks when the lost block must be reconstructed. Thus, each original disk retrieval block is read twice during two different service rounds. Therefore, the scheduler must know when to read the remaining $(D-1)$ blocks in order to reconstruct the lost block: We assume that disk d_f fails during the service round T_f when disk retrieval block b_f is being retrieved. The scheduler must be able to locate the disk d from which the original disk retrieval block b_f is currently retrieved, and the parity group g that block b_f belongs to. `procedure R_disk_group(`b_c`,` d_c`,` g_c`)` calculates for a given b_c (or T_c) its corresponding disk d_c and parity group g_c.

```
procedure R_disk_group($b_c$, $d_c$, $g_c$)
/* Input parameter */
    /* $b_c$:  current disk retrieval block number.  It corresponds to the current
    service round $T_c$ */

/* Output parameters */
    /* $d_c$:  current disk */
    /* $g_c$:  current parity group */

BEGIN
    $g = b_c$ div $(D-1)$ ; /* the line */
    $d = b_c$ mod $(D-1)$ ; /* the column */

    IF $d == 0$ $\{$ /* the last original disk retrieval block in the parity group */
       $g_c = g$;
       $d_c = (D-1)$;
       IF ($g$ mod $D \neq 0$) $\{$
          $d_c = D$ $\}$ $\}$
    ELSE $\{$ /* $d \neq 0$ */
       $g_c = g + 1$;
       IF ($d$ < ($g_c$ mod $D$)) $\{$ /* left of diagonal */
          $d_c = d$; $\}$
       ELSEIF ($d$ > ($g_c$ mod $D$)) $\{$ /* right of diagonal */
          $d_c = d + 1$; $\}$ $\}$
END
```

Now we calculate the service round numbers at which the reconstruction of a lost disk retrieval block stored on the failed disk d_f must be carried out. Only the *original* blocks of disk d_f must be reconstructed. Thus, for the parity group numbers [d_f, $D + d_f$,..., $((j \cdot D) + d_f)$,...] no reconstruction is needed, since disk d_f stores the *parity* disk blocks of these parity groups.

`procedure R_sec_block_time(`d_f`,` b_f`,` g_c`,` b_r`,` T_r`)` delivers the block number b_r of the *lost and needed* disk retrieval block belonging to the parity group g_c and the expected time (scheduling round) T_r of its reconstruction.

```
procedure R_rec_block_time($d_f$, $b_f$, $g_c$, $b_r$, $T_r$)
/* Input parameters */
    /* $d_f$:  The failed disk */
    /* $b_f$:  The sequence number of the disk retrieval block read, when $d_f$
    fails.  $b_f$ corresponds to the service round $T_f$ */
    /* $g_c$:  The current parity group number */

/* Output parameters */
    /* $b_r$:  The sequence number of a lost and needed disk retrieval
    block belonging to parity group $g_c$ */
    /* $T_r$:  the scheduling round where to reconstruct $b_r$ in parity group $g_c$ */

BEGIN
    R_disk_group($b_f$, $d$, $g$); /* returns $d$ and $g$ */
    $k$ = $g$ mod $D$;     /* $g = z \cdot D + k$ */

    $h$ = $g_c$ mod $D$;
    $w$ = $g_c$ div $D$;     /* $g_c = w \cdot D + h$ */

    IF ($h$   ==   $d_f$) /* No reconstruction is needed, since a parity block was
    stored on disk $d_f$ */

    ELSE { /* $h \neq d_f$ */
       $b_r = (h-1) \cdot (D-1) + d_f + w \cdot D \cdot (D-1)$;
       IF ($1 \leq h \leq (d_f - 1)$) { /* right of the diagonal */
          $b_r = b_r - 1$; }

       $T = T_f + (d_f - d)$;
       IF ($d \leq k < d_f$) {
          $T = T - 1$; }

       $T_r = T + (g_c - g) \cdot (D-1)$;
       IF ($h > d_f$) { /* left of diagonal */
          $T_r = T_r + 1$; } } /* end ELSE */
END
```

Using R_rec_block_time(d_f, b_f, g_c, b_r, T_r), the disk scheduler can compute exactly when a reconstruction is required and which $(D-2)$ original blocks and parity block are required to decode the lost disk retrieval block for the parity group g_c.

B.2 Mirroring-Based Reliability

We assume one video object to be stored and replicated on the video server. The storage of the *original* disk retrieval blocks follows the CGS algorithm. As for the parity-based scheme, we assume a video object containing $Z \cdot D \cdot (D-1)$ original disk retrieval blocks. The storage of the *replicated* disk retrieval blocks is round robin in order to distribute the load of a failed disk over as many disks as possible. A disk d_i contains $Z \cdot (D-1)$ *original* disk retrieval blocks and additionally Z *replicated* disk retrieval blocks from each of the other disks d_1,..,d_{i-1},d_{i+1},..,d_D. Each disk is assumed to be partitioned into two parts. The first part contains original data and the second one stores replicated data. Figure B.4 shows the storage layout of original disk retrieval blocks of one video object, and how the original blocks of disk d_i are replicated over the remaining disks.

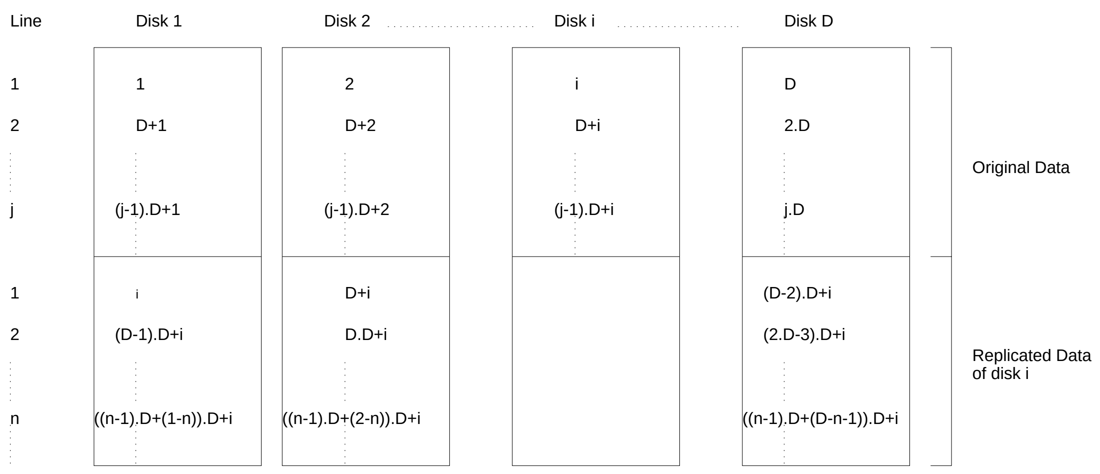


Figure B.4: Mirroring data layout for one video object stored on a video server with D disks.

In section 3.2, we used for RAID 5 the notation of *parity group* to determine the retrieval line within the matrix. For mirroring we use the term **retrieval line**. As in section 3.2, we describe the scheduling order of video data during normal operation and single disk failure operation mode. We consider one video object that is replicated within the server. In the following discussion, we describe how to replicate the content (original blocks) of a disk d_i. `procedure M_repl_blocks(`d_i, d_j, n, r_n`)` delivers the disk retrieval block numbers of the original blocks stored on disk d_i that have their replicas on disk d_j. This procedure is derived from the data layout order shown in Figure B.4.

```
procedure M_repl_blocks($d_i$, $d_j$, $n$, $r_n$)
/* Input parameters */
    /* $d_i$:  Disk, where original disk retrieval blocks with numbers $i$,
    $D+i$,..,$(j-1) \cdot D+i$,..  are stored */
    /* $d_j$:  Disk that stores some replicas of the original blocks of $d_i$, where
    $j \neq i$ */
    /* $n$:  the number of the replica on disk $d_j$, $n \in [1..Z]$ */

/* Output parameter */
    /* $r_n$:  Sequence number of the $n^{th}$ replica of $d_i$ on $d_j$ */

BEGIN
    $r_n = ((n-1) \cdot D + (d_j - n)) \cdot D + d_i$;
    IF ($d_j > d_i$) { /* right side of disk $d_i$ */
       $r_n = r_n - D$; }
END
```

B.2.1 Scheduling and Retrieval for Normal Operation Mode

During normal operation mode, only original data is retrieved. The scheduler retrieves consecutive *original* disk retrieval blocks from consecutive disks. Using (V_{ws},S_{ss}), one disk retrieval block is retrieved at each service round. At a service round, the scheduler retrieves a disk retrieval block from disk d. During the next service round, a disk retrieval block is retrieved from disk ($(d+1)$ mod D). The scheduler of disk d_i must know when it should retrieve an original disk retrieval block. This is resolved using the `procedure M_disk_line(`b_c`, `d_c`, `l_c`)` that delivers the disk d_c that must deliver an original block having the number b_c. It also delivers the retrieval line l_c that will be needed later when we discuss the single disk failure operation mode. Note that b_c corresponds to service round T_c.

```
procedure M_disk_line($b_c$, $d_c$, $l_c$)
/* Input parameter */
    /* $b_c$:  Current disk retrieval block number.  It corresponds to
    the current service round number $T_c$ */

/* Output parameters */
    /* $d_c$:  Current disk */
    /* $l_c$:  Current retrieval line */

BEGIN
    $l_c = ((b_c - 1)$ div $D) + 1$;
    $d_c = b_c - ((l_c - 1) \cdot D)$;
END
```

B.2.2 Scheduling and Retrieval for Single Disk Failure Operation Mode

During single disk failure mode, each *lost and needed original* disk retrieval block must be reconstructed using its replica. Therefore, each disk scheduler must know when to read which replicated data.

Assume disk d_f fails when the disk retrieval block b_f is being retrieved during the service round T_f. The scheduler must determine the disk d from which the original disk retrieval block b_f was retrieved and the retrieval line l of this block. `procedure M_disk_line(`b_f`, `d`, `l`)` will be used to get the expected d and l.

We now calculate the service rounds numbers at which the replica of a disk retrieval block stored on the failed disk d_f must be retrieved. The solution of this problem is given by `procedure M_rec_block_time(`d_f`, `b_f`, `l_c`, `b_r`, `T_r`, `d_r`)` as:

```
procedure M_rec_block_time($d_f$, $b_f$, $l_c$, $b_r$, $T_r$, $d_r$)
/* Input parameters */
    /* $d_f$:  The failed disk */
    /* $b_f$:  The disk retrieval block read, when $d_f$ fails */
    /* $b_f$ corresponds to $T_f$ */
    /* $l_c$:  The current retrieval line */

/* Output parameters */
    /* $b_r$:  The number of the original block that must be reconstructed for $l_c$
    */
    /* $T_r$:  The service round, at which the replica of the disk retrieval
    block with number $b_r$ must be retrieved */
    /* $d_r$:  The disk that contains the replica of the disk retrieval block with
    number $b_r$ */

BEGIN
    disk_line($b_f$,$d$,$l_f$); /* delivers $d$ and $l_f$ */
    $T = T_f + (d_f - d)$; /* time to first reconstruction if $d \leq d_f$ */
    $T_r = T + (l_c - l_f) \cdot D$;
    $b_r = (l_c - 1) \cdot D + d_f$;
    $b = (l_c - 1)$ mod $(D - 1)$; /* disk of the replica */
    $d_r = b + 1$;
    IF ($d_r \geq d_f$) $\{$ /* right side of disk $d_f$ */
       $d_r = d_r + 1$; $\}$
END
```

`procedure M_rec_block_time(`d_f, b_f, l_c, b_r, T_r, d_r`)` delivers (i) the number b_r of the *lost and needed* disk retrieval block belonging to the current retrieval line l_c, (ii) the expected time T_r, at which the replica of the disk retrieval block with number b_r must be retrieved, and (iii) the disk d_r that contains the replica.

Appendix C

Sommaire détaillé en français

Nous présenterons dans cet appendice un sommaire détaillé en fran̦cais du travail effectué dans cette thèse [Gafsi 99].

C.1 Introduction

La puissance des ordinateurs sans cesse accrue et l'évolution continue des technologies des réseaux et de stockage ont donné naissance à de nouvelles applications qui supportent les données multimédia: la vidéo et l'audio, qui sont commément appelées **applications multimédia**. Malgré l'amélioration considérable de la capacité de stockage et de la bande passante des systèmes informatiques, l'information multimédia reste premièrement trop volumineuse, ce qui nécessite sa compression et deuxièmement gourmande en bande passante, ce qui requiert la présence de réseaux haut débit et de systèmes puissants pour la traiter. D'autre part, l'information multimédia doit être souvent transmise, traitée et présentée dans des délais fixes. Toutes ces caractéristiques poussent à con̦cevoir de nouveaux systèmes de communication pour mettre en oeuvre les applications multimédia. Ces nouveaux systèmes, appelés aussi systèmes multimédia, peuvent être des systèmes de *création*, des systèmes de *traitement* ou encore des systèmes de *stockage* de l'information multimédia. Nous nous limitons dans ce chapitre aux systèmes de stockage de l'information multimédia dont le **serveur vidéo** présente un exemple typique.

Un serveur vidéo est un système de stockage et de livraison de la vidéo qui offre typiquement le service de la **vidéo à la demande (VOD)**. Le serveur vidéo stocke une large collection de vidéos. L'utilisateur accède au serveur vidéo à travers un réseau et re̦coit la vidéo qu'il a choisie. Ainsi, il n'a pas besoin de quitter son domicile pour regarder sa vidéo préférée. En outre, il peut regarder sa vidéo favorite quand il le désire et a la possibilité d'interagir en utilisant les commandes d'interactivité classiques telles que *pause*, *fast-forward*, *fast-backward*, etc.. La VOD permet aussi à un nombre très large d'utilisateurs d'accéder *simultanément* à la même vidéo où chacun peut se servir des commandes d'interactivité indépendamment des autres utilisateurs.

Les principaux critères de performance d'un serveur vidéo sont le nombre de flux qu'il peut supporter simultanément et le coût total de son architecture. La conception d'un serveur vidéo doit donc admettre le plus possible de flux pour une configuration -et donc un coût- donnés. Avant de se pencher sur les détails de la conception du serveur vidéo, nous introduirons comme suit l'environnement de ce dernier: Puisque le dispositif de stockage constitue la composante la plus importante du serveur vidéo, nous discuterons d'abord le choix du dispositif de stockage qui répond le mieux aux contraintes du serveur vidéo en fait de volume de stockage, débit de transfert, temps d'accès ainsi que coût du dispositif. Ensuite, nous citerons les divers critères de performance et de qualité de service du serveur vidéo. Ces critères seront utilisés pour évaluer et comparer les différents algorithmes et mécanismes que nous allons considérer pour le stockage, l'ordonnancement, l'extraction des données, ou bien pour assurer la fiabilité du serveur vidéo.

C.1.1 Serveurs vidéo: environnement et ressources

Le serveur vidéo qui est un système permettant le stockage et la livraison des données vidéo impose des contraintes qui sont liées à la nature de l'information vidéo. En effet, cette dernière – même sous sa forme compressée – est volumineuse, gourmande en bande passante et doit être traitée dans des délais fixes. Par conséquent, la conception et la réalisation d'un serveur vidéo doivent satisfaire les critères suivants [Chung 96]:

- Afin de servir une large population de clients simultanément, la capacité de stockage (*storage capacity*) ainsi que le débit (*transfer rate*) doivent être élevés.
- Pour offrir à chaque client un service ininterrompu, le serveur vidéo doit intégrer des mécanismes de contrôle d'admission et d'ordonnancement des différents flux.
- La conception du serveur vidéo et les algorithmes utilisés pour sa réalisation doivent être adaptés à l'augmentation de l'échelle (*scalable*) en nombre de vidéos stockées et de clients.

Stockage de la vidéo

Commençons par un exemple qui donne une idée du besoin du serveur vidéo en matière de capacité de stockage et de bande passante: supposons que le serveur vidéo stocke 2000 vidéos, chacune d'une durée de 100 minutes et codée en MPEG-2 à $8 \mathrm{Mbit/sec}$. Admettons aussi que le serveur vidéo doive servir 1000 clients simultanément. La demande en capacité de stockage est par conséquent de l'ordre de $12 \mathrm{TByte}$ et la demande en débit est de $8 \mathrm{Gbit/sec}$. Il faut que d'une part le dispositif de stockage que le serveur vidéo utilise réponde à ces besoins en capacité et en débit et que, d'autre part le coût du serveur vidéo soit maintenu le plus bas possible afin d'assurer sa rentabilité. Plusieurs dispositifs de stockage sont candidats pour le serveur vidéo: les cassettes (*tape*), les disques magnétiques (*magnetic disks*), les disques optiques (*optical disks*), les disques vidéo digitaux (*DVD-ROM*) et la mémoire vive (*RAM*). Un résumé des différentes caractéristiques de ces dispositifs est donné dans TAB C.1 [Lu 96].

Dispositif	Débit ($\mathrm{Mbit/sec}$)	Coût/GByte	Problèmes
Cassettes	0.8	0.1	débit faible et accès séquentiel
Disques optiques	2.4	250	débit faible
Disques vidéo digitaux	10	50	débit faible et capacité réduite
Disques magnétiques	80	400	coût élevé
mémoire vive	800	10000	coût trop élevé

Table C.1: Dispositifs de stockage (prix env. 1996)

D'après TAB C.1, les disques magnétiques sont le dispositif le plus adapté au stockage de l'information vidéo au sein d'un serveur vidéo. En effet, un disque magnétique:

- a une grande capacité de stockage qui peut atteindre plusieurs GBytes, contrairement au disque vidéo digital.
- a un débit élevé de l'ordre de dizaines de $\mathrm{Mbit/sec}$ contrairement à la cassette, au disque optique et au disque vidéo digital.
- permet l'accès rapide et aléatoire à l'information vidéo contrairement à la cassette et au disque optique.
- a un coût faible comparé à celui de la mémoire vive.

Afin d'optimiser le coût d'un serveur vidéo, des travaux tels que [Doganata 96, Shastri 98, KIENZ 95, WILK 96] ont proposé des architectures de serveurs vidéo basées sur le **stockage hiérarchique**: les vidéos populaires sont stockées dans les disques magnétiques ou même dans la mémoire vive pour les plus populaires d'entre elles. Les vidéos peu populaires quant à elles sont *archivées* dans les cassettes ou même dans les disques optiques et disques vidéo digitaux. Le changement de la popularité d'une vidéo la déplace d'un niveau à l'autre dans la structure hiérarchique.

De nombreux autres travaux ont proposé des serveurs vidéo utilisant les disques magnétiques comme unique dispositif de stockage de la vidéo. Cette proposition se justifie surtout par la baisse continue du prix des disques magnétiques. En effet, le coût d'un MByte d'un disque magnétique ne cesse de diminuer (40% par an), et ce depuis presque vingt ans [Grochowski 97a].

Parallèlement à la baisse du prix des disques magnétiques, leur capacité de stockage ne cesse de grimper (25% à 50% par an) pour dépasser aujourd'hui $18 \ \mathrm{GByte}$. Un autre argument en faveur du choix des disques magnétiques est leur débit de transfert qui a connu au cours des dix dernières années une augmentation annuelle de 40%. Ce débit peut atteindre actuellement $40 \ \mathrm{MByte/sec}$ [Grochowski 97c]. En revanche, les disques magnétiques connaissent une réduction faible de leur temps d'accès. Ce temps d'accès, qui comprend le délai de positionnement sur la bonne piste et le délai de rotation, ne se réduit que de 5% par an. La valeur moyenne du temps d'accès est actuellement de l'ordre de $10\mathrm{ms}$. Nous discuterons ultérieurement l'impact du temps d'accès au niveau du disque magnétique sur la performance du serveur vidéo.

Caractéristiques des disques magnétiques

Afin d'évaluer la performance du serveur vidéo, une connaissance des caractéristiques et performances de ses composantes est indispensable. Nous avons opté pour les disques magnétiques en tant que dispositif de stockage des données vidéo. Cette section décrit ses principales caractéristiques.

Un disque magnétique est constitué typiquement de plusieurs [≈ 20] **surfaces** (*platters*) qui effectuent des mouvements de rotation autour d'un **fuseau central** (*spindle*). Chaque surface possède sa propre **tête** (*disk head*) responsable de la lecture des données. Une surface est composée d'un grand nombre [≈ 500] de cercles concentriques que nous appelons **pistes** (*tracks*). Chaque piste est composée de blocs physiques de taille fixe de l'ordre de $256\mathrm{Bytes}$ ou $512\mathrm{Bytes}$. Les pistes qui appartiennent à des surfaces différentes et qui ont la même distance au fuseau central forment ensemble un **cylindre** (*cylinder*).

Pour extraire un bloc, un disque parcourt deux phases: la première est la phase de localisation du bloc où la tête du disque doit être ajustée et la deuxième représente la phase où les données sont effectivement lues. La phase de localisation du bloc contient principalement un **délai de positionnement sur la bonne piste** (*seek time*) et un **délai de rotation** (*rotational latency*). Le délai de positionnement sur la bonne piste est le temps nécessaire pour déplacer la tête du disque et la remettre sur la piste contenant le bloc à extraire. Le délai de rotation est le temps nécessaire au sein d'une piste pour repositionner la tête du disque sur le début du bloc à extraire.

La Figure C.1 présente les différentes caractéristiques d'un disque magnétique. Notez que seules les opérations de lecture des données sont des opérations utiles pour un disque, et le pourcentage du temps utile pour un disque dépend directement du délai de positionnement sur la bonne piste et du délai de rotation. En effet, pendant les opérations de positionnement sur la bonne piste et de rotation, le disque magnétique n'est pas capable d'effectuer des transferts de données. Par conséquent, une bonne utilisation du disque exige la réduction des délais de positionnement sur la bonne piste et de rotation. Pour avoir plus de détail sur les disques magnétiques, nous invitons le lecteur à recourir à [WILK 94, WORT 95, Hennessy 90, Mr.X].

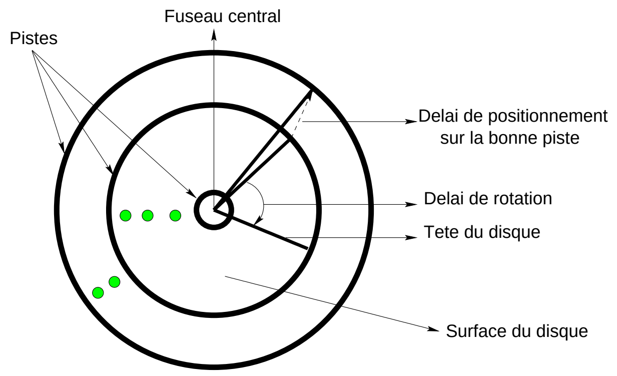


Figure C.1: Caractéristiques d'un disque magnétique

Environnement du serveur vidéo

La Figure C.2 illustre l'environnement d'un serveur vidéo. Elle distingue trois parties: le serveur vidéo, un réseau haut débit et les clients.

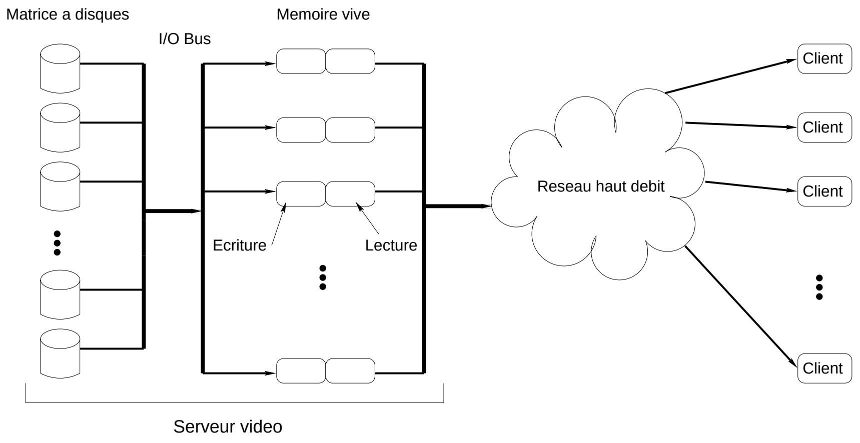


Figure C.2: Architecture et environnement du serveur vidéo

Le serveur vidéo contient principalement trois composantes:

- Une matrice à disques pour le stockage de la vidéo: Afin d'atteindre une grande capacité de stockage et une large bande passante, le serveur vidéo est typiquement composé de plusieurs disques magnétiques. Un système qui contient plusieurs disques magnétiques est nommé une **matrice à disques** (*disk array*). Cette matrice à disques est composée de disques multiples stockant les vidéos. La capacité de stockage de la matrice est donc proportionnelle au nombre de disques qu'elle contient. Notez que chaque vidéo est découpée en morceaux appelés **blocs**. Ceux-ci sont ensuite stockés sur une ou plusieurs composantes du serveur vidéo. La méthode de distribution des blocs d'une vidéo fait l'objet de la section C.2.6.

- Un bus entrée/sortie (I/O bus) (e.g. SCSI) qui sert à transférer les données extraites des disques vers le buffer. Afin d'exploiter toute la bande passante de la matrice à disques, la capacité de transfert du I/O bus doit être supérieure à la somme des débits de transfert de tous les disques de la matrice.

- Une mémoire tampon (buffer): Le débit de transfert des données au niveau d'un disque magnétique atteint aujourd'hui des dizaines de $\mathrm{Mbit/sec}$. Ce débit dépasse largement le débit de consommation du flux vidéo chez le client (jusqu'à $8\ \mathrm{Mbit/sec}$ pour MPEG-2). Afin d'absorber cette différence de débit et assurer un flux vidéo continu, les blocs vidéo extraits de chaque disque sont stockés temporairement au niveau du buffer avant d'être envoyés vers le client correspondant. Comme indiqué dans la Figure C.2, le buffer est fondé sur la méthode du double buffer (*double buffering*): le serveur vidéo réserve pour chaque flux deux espaces buffer. Chaque espace a la taille d'un bloc et sert soit à l'écriture d'un bloc provenant du I/O bus, soit à la lecture d'un bloc avant son envoi vers le client. Notez que les deux espaces buffer alternent leurs rôles d'une unité de service à une autre.

Un réseau haut débit est indispensable pour véhiculer des flots de données à haut débit. En effet, même sous sa forme compressée, l'information vidéo reste très gourmande en bande passante (jusqu'à $8\mathrm{Mbit/sec}$ pour MPEG-2). Actuellement, la plupart des réseaux ne sont pas capables de fournir la bande passante nécessaire pour permettre à un serveur vidéo de servir des centaines ou des milliers de clients simultanément. Cette limitation en matière de bande passante au niveau du réseau est le facteur décisif qui empêche le marché de la VOD d'atteindre une large population.

Chaque client est connecté au serveur vidéo via le réseau haut débit. Il s'agit bien d'un rapport de producteur-consommateur. En effet, le client envoie sa requête au serveur vidéo. L'entité centrale du serveur vidéo, appelée aussi **meta serveur**, fournit à ce nouveau client la liste des vidéos disponibles. Le client renvoie alors une requête indiquant la vidéo qu'il a choisie. Ensuite, le meta serveur effectue un contrôle d'admission de ce nouveau client. Dans le cas où ce dernier est admis, le meta serveur informe le serveur vidéo de la présence de ce client, qui, à son tour, établit une connection avec ce dernier. Le client commence ainsi à recevoir son propre flux vidéo en temps réel. Le client dispose des fonctions classiques d'interactivité et a la possibilité d'envoyer des requêtes telles que *fast forward* ou *pause* au serveur vidéo, qui, de son côté, traite cette demande et exécute la fonction désirée par le client.

C.1.2 Critères de performance et de qualité de service du serveur vidéo

Les critères de performance du serveur vidéo sont le **débit** du serveur vidéo, le **besoin en buffer** et le **coût** du serveur vidéo. Les critères de qualité de service du serveur vidéo sont sa **fiabilité**, le **temps de latence initial** ainsi que la **répartition de la charge** entre ses composantes.

Les ressources du serveur vidéo sont limitées (débit au niveau des disques, I/O bus, taille du buffer,..) et par conséquent le serveur vidéo ne peut admettre qu'un nombre limité de clients. Par définition, le débit du serveur vidéo est le nombre maximum de clients qui peuvent être admis simultanément. Avant d'admettre un nouveau client, le serveur vidéo effectue un **contrôle d'admission**. Si le nouveau client est admis, une quantité de ressources lui est allouée afin d'assurer un service ininterrompu.

Le buffer est une partie très coûteuse du serveur vidéo et le besoin en buffer affectera donc le coût du serveur vidéo. Nous verrons dans ce chapitre comment le besoin en buffer peut être décisif dans le choix des différents algorithmes qu'implémente le serveur vidéo.

La durée moyenne de fonctionnement avant défaillance (*Mean Time To Failure (MTTF)*) d'un disque est de l'ordre de 300000 heures. Le serveur vidéo contient typiquement des *centaines* de disques fondé sur la matrice à disques afin de répondre aux demandes en capacité de stockage et en bande passante. Ceci implique que la MTTF du serveur vidéo est beaucoup plus faible que celle d'un seul disque. D'où la nécessité de concevoir des mécanismes de fiabilité au sein du serveur vidéo pour le protéger des pannes de ses composantes. Dans le contexte des systèmes de fichiers classiques, plusieurs variations de matrices à disques incluant des mécanismes de fiabilité ont été proposées; et elles sont connues sous le nom **RAID** (*Redundant Array of Inexpensive Disks*) [Patterson 88]. Six niveaux de RAID ont été distingués, qui dépendent du mécanisme de fiabilité utilisé et aussi de la granularité de répartition de l'information à travers les différents disques de la matrice. RAID sous sa forme standard (hardware RAID) n'est pas adapté aux applications multimédia telle que la vidéo à la demande sous la réserve où il ne répond pas aux contraintes temps réel de ces applications. Ceci pousse à concevoir des mécanismes de fiabilité au niveau de l'application pour rendre le serveur vidéo tolérant aux pannes. Nous allons considérer deux mécanismes de fiabilité qui sont basés sur la **redondance**: le premier concerne la simple **réplication** des données originales (*mirroring* qui est analogique à *RAID1*), tandis que le deuxième mécanisme concerne la **parité** (*parity* qui est analogique à *RAID2-RAID6*).

Le concepteur d'un serveur vidéo doit non seulement optimiser le débit et assurer la fiabilité, mais aussi veiller à ce que les solutions proposées soient économiquement viables. Étant donnée l'interdépendance du coût total du serveur vidéo et du débit qu'il peut atteindre, une évaluation précise du serveur vidéo doit considérer le coût par flux, qui est le coût total du serveur vidéo divisé par le nombre maximal des flux admis.

Le temps de latence est le temps écoulé entre l'instant où le client envoie sa requête au serveur vidéo et l'instant où la première trame arrive au client. Nous supposons que le temps de latence initial ne concerne que les délais dus au traitement de la requête du côté du serveur vidéo et nous admettons donc que le réseau ne cause aucun délai supplémentaire.

Il est souhaitable pour le serveur vidéo que la charge soit répartie équitablement entre ses différentes composantes. Notez qu'une répartition inéquitable de la charge peut, dans certains cas, entraîner la réduction du débit du serveur vidéo ou augmenter le temps de latence initial pour certains clients. Nous verrons plus tard comment la technique de répartition de la vidéo influence la performance et la répartition de la charge ainsi que le fiabilité du serveur vidéo.

C.2 Conception des serveurs vidéo

C.2.1 Motivation

La conception d'un serveur vidéo doit satisfaire à deux critères majeurs: d'une part une performance élevée du serveur vidéo en nombre de clients admis et d'autre part un coût bas par client servi. Le deuxième critère impose l'utilisation de hardware standard pour la réalisation du serveur vidéo. En effet, l'utilisation des machines très puissantes est une solution trop coûteuse, et donc non rentable. L'utilisation d'un matériel (hardware) standard permet la conception d'un serveur vidéo à la fois flexible, robuste à la largeur de l'échelle (scalable) et surtout bénéficiant des améliorations sans cesse accrues de la performance de la hardware. Cependant, une telle décision implique de réaliser toute l'intelligence du serveur vidéo en software. En d'autre termes, toutes les fonctions du serveur vidéo, telles que la répartition des données ou encore la fiabilité du serveur, sont gérées par l'application et non dédiées à la hardware. Nous détaillerons maintenant les différentes étapes de la conception d'un serveur vidéo illustrées dans la Figure C.3.

La première étape est de définir la configuration et l'architecture du serveur vidéo. Une alternative est à envisager: soit on opte pour une architecture centralisée, où le serveur vidéo est constitué d'une machine unique contenant une matrice à disques, soit on considère un serveur vidéo distribué contenant plusieurs machines. Nous allons discuter en détail cette deuxième possibilité dans la section C.2.2 où nous allons argumenter notre choix d'une architecture distribuée du serveur vidéo.

Une seconde étape est de définir l'algorithme de répartition des données sur le serveur vidéo. Il faut noter que répartir la vidéo sur le serveur vidéo ne veut pas dire répliquer la vidéo, mais découper celle-ci en morceaux, appelés blocs, et stocker les différents blocs de la vidéo dans le serveur vidéo. La manière de stocker ces blocs déterminera la méthode de répartition des données. Une analyse et comparaison des différents algorithmes de répartition des données seront détaillées dans la section C.2.6.

Les étapes 3, 4 et 5 de la Figure C.3 ont pour but de définir les techniques d'extraction des

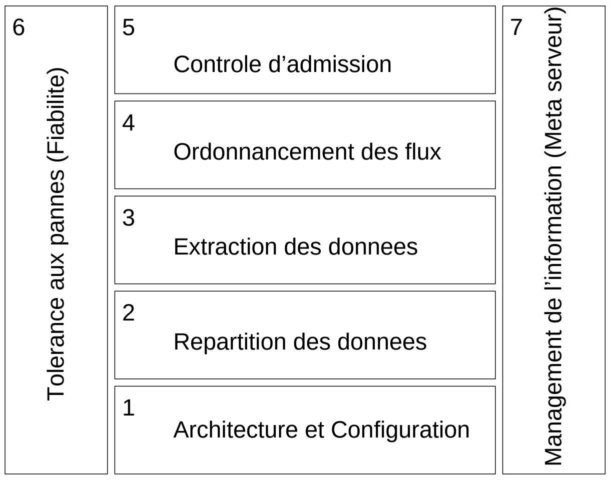

6
Tolerance aux pannes (Fiabilite)
5
Controle d'admission
4
Ordonnancement des flux
3
Extraction des donnees
2
Repartition des donnees
1
Architecture et Configuration
7
Management de l'information (Meta serveur)

Figure C.3: Les différentes étapes dans la conception d'un serveur vidéo.

données (voir section C.2.3), d'ordonnancement des flux multiples (voir section C.2.4) et de contrôle d'admission des nouveaux flux (voir section C.2.5). Ces trois étapes sont liées comme nous les verrons ultérieurement.

L'étape 6 de la Figure C.3 représente l'intégration de la fiabilité au sein du serveur vidéo afin de le rendre tolérant aux pannes de ses composantes. Cette étape, comme nous l'avons présenté dans la Figure C.3, influe sur toutes les étapes précédentes. Ainsi, la conception de l'architecture doit prévoir le surcoût que la fiabilité implique en terme de volume de stockage et de bande passante supplémentaires. La technique de répartition des données doit aussi gérer l'information redondante et la répartir. L'ordonnancement et l'extraction des flux doivent, pour leur part, réagir à des pannes. Enfin, le mécanisme de contrôle d'admission doit prévoir le cas des pannes et réserver la bande passante nécessaire pour satisfaire tous les clients admis durant le mode de défaillance.

Une étape très importante dans la conception du serveur vidéo est de définir la fonction du meta serveur qui représente le coeur de tout le système (étape 7 dans la figure). Le meta serveur est celui qui gère l'interaction entre les différentes composantes du serveur vidéo et aussi entre le serveur vidéo et le client. C'est uniquement le meta serveur qui a l'information sur la configuration complète du serveur vidéo ainsi que sur les techniques de répartition des données utilisées. Le contrôle d'admission se fait également au niveau du meta serveur.

Dans le reste de cet article, nous traiterons les étapes de conception 1 à 6 qui sont présentées dans la Figure C.3. Pour chaque étape, nous choisirons la méthode ou l'algorithme que notre serveur vidéo utilisera. Notez que chaque étape est fondée sur les choix qui ont été pris dans les étapes qui la précédent.

C.2.2 Architecture et configuration: la matrice à serveur

Les serveurs vidéo sont des systèmes où les solutions distribuées sont essentielles au vu du volume très large des données à stocker et du débit très haut à atteindre. Nous avons proposé une architecture de serveur vidéo distribuée dont voici les caractéristiques: étant donnée la demande en matière de capacité de stockage et de bande passante, le serveur vidéo est typiquement basé sur la *matrice à disques* appelée aussi **noeud**. En outre, pour pouvoir servir plus de clients, le serveur vidéo doit voir sa capacité en terme de bande passante augmenter. Cependant, cette demande en bande passante ne peut pas être satisfaite en ajoutant simplement des disques au sein du même noeud. En effet, un noeud, à cause des limitations imposées par ses composantes (buffer, CPU, I/O bus, etc.), ne peut contenir qu'un nombre restreint de disques. Par conséquent, augmenter la capacité du serveur vidéo pour admettre plus de clients revient à ajouter de nouveaux noeuds à côté des noeuds déjà existants.

Traditionnellement, un serveur vidéo est composé de plusieurs noeuds *indépendants* et *autonomes*: chaque vidéo est entièrement stockée sur un des noeuds. Cette configuration de *noeuds autonomes* présente deux inconvénients majeurs:

1. La charge du serveur vidéo est distribuée d'une façon inégale entre les différents noeuds. Les noeuds qui contiennent les vidéos les plus populaires sont **surchargés** (*hot spots*) tandis que d'autres noeuds restent sous-chargés.

2. Le nombre de clients accédant à une vidéo très populaire est limité par la capacité du noeud où cette vidéo est stockée. Afin de permettre à plus de clients d'accéder à la même vidéo populaire, cette dernière doit être *copiée* sur d'autres noeuds. Mais, ceci est inefficace à l'égard de la capacité de stockage du serveur vidéo. En outre, les variations de popularité des différentes vidéos stockées rendra la mise à jour du contenu du serveur vidéo complexe.

Notre approche s'inspire de la configuration de la matrice à disques où chaque vidéo est distribuée sur les différents disques de cette matrice. Nous avons ainsi proposé une architecture de serveur vidéo où plusieurs noeuds constituent ensemble une matrice appelée la **matrice à serveurs** [BERN 96a]. La Figure C.4 illustre l'architecture de la matrice à serveurs.

Chaque noeud de la matrice à serveurs est lui-même une matrice à disques. Contrairement aux serveurs autonomes, un noeud de la matrice à serveurs ne stocke qu'une partie de chaque vidéo. En effet, chaque vidéo est découpée en *blocs* et ces derniers sont répartis sur les différents noeuds de la matrice à serveurs. Par exemple, le premier bloc de la vidéo est stocké sur le premier noeud, le deuxième bloc sur le deuxième noeud, $\cdots$, le i ème bloc est stocké sur le noeud $(i \bmod N) + 1$ où N représente le nombre de noeuds qui sont contenus dans le serveur vidéo. La répartition de la charge entre les noeuds du serveur est équitable et ne dépend pas

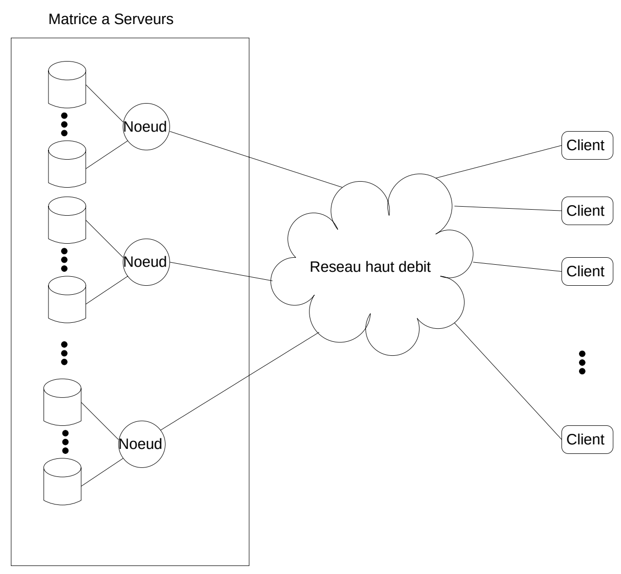


Figure C.4: La Matrice à serveurs

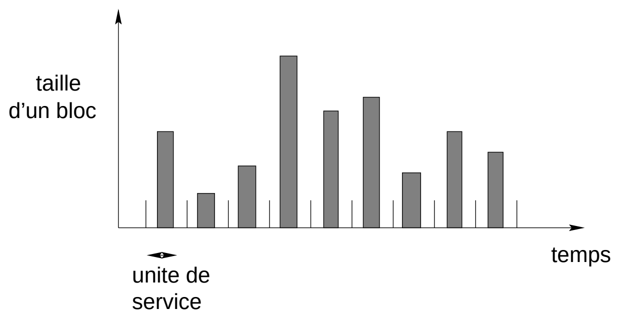
de la popularité des vidéos consommées. Cette répartition de la charge est même *parfaite* si chaque vidéo est répartie sur tous les noeuds de la matrice à serveurs. De plus, la capacité totale du serveur vidéo peut être utilisée pour servir un grand nombre de clients, qui, tous, peuvent utiliser la même vidéo très populaire. Cette répartition a l'avantage d'éviter de *copier* les vidéos populaires sur plusieurs noeuds.

L'architecture de la matrice à serveurs indique que chaque vidéo est découpée en plusieurs blocs et que ces derniers sont stockés sur les différents noeuds de la matrice à serveurs. En revanche, cette architecture ne précise ni (i) la fréquence d'extraction des données, ni (ii) l'ordre suivant lequel les différents flux sont servis par les noeuds et des disques du serveur vidéo ni (iii) la granularité de distribution des données vidéo sur les noeuds ainsi que sur les disques à l'intérieur d'un noeud. Les sections C.2.3, C.2.4 et C.2.6 examineront en détail ces aspects.

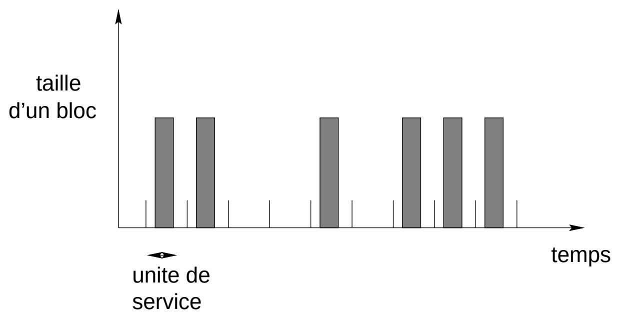
C.2.3 Extraction des données (Data Retrieval)

Par définition, le mécanisme d'extraction détermine la fréquence avec laquelle les différents blocs appartenant au même flux sont lus au niveau du serveur vidéo et ensuite délivrés vers le client correspondant. Considérons les techniques d'extractions périodiques (*round-based data retrieval*) où le temps de service est découpé en intervalles fixes et égaux appelés **unités de service**. Chaque flux est ainsi servi une fois durant une unité de service. Afin de bien présenter les différentes techniques d'extractions, une connaissance de la composition d'une vidéo nous semble essentielle. Une vidéo est composée de centaines ou milliers de trames. Pour des raisons d'efficacité de stockage et afin de réduire la bande passante nécessaire, la vidéo est compressée avant d'être stockée dans le serveur vidéo. Chaque vidéo a un débit fixe en terme de $trames/sec$, mais la taille des trames diffère suivant la méthode de compression adoptée. Généralement, deux options sont utilisées pour la compression de la vidéo. La première option génère un flux à qualité constante mais ayant un débit variable (*VBR: Variable Bit Rate*); une vidéo codée en VBR est composée de trames de tailles différentes. La deuxième option génère un flux ayant un débit contrôlé et constant, mais une qualité variable (*CBR: Constant Bit Rate*); une vidéo codée en CBR est composée de trames de la même taille. L'avantage de VBR par rapport à CBR est sa qualité constante. Cependant, CBR permet le stockage de la vidéo en blocs de même taille et aussi l'extraction de ces différents blocs avec une périodicité façile à gérer, ce qui n'est pas le cas pour VBR.

Pour VBR, il faut décider lors du découpage et répartition de la vidéo codée VBR, si les blocs à stocker auront la même taille ou des tailles différentes. Cette décision définira la technique d'extraction des blocs appartenant au même flux VBR. Principalement, deux méthodes se distinguent dans ce contexte: la première méthode est appelée **CDL** (*Constant Data Length*); la deuxième est **CTL** (*Constant Time length*) [CHA 94, CHAN 96]. La Figure C.5 illustre ces deux méthodes pour un flux. Figure C.5(a) montre que pour CTL, les blocs à extraire ont des tailles variables et durant chaque unité de service, un bloc est extrait du serveur vidéo. Dans le cas de CTL, chaque flux demande un nombre d'accès aux disque qui est égal au nombre d'unités de service qui correspondent à la durée de la vidéo consommée par ce flux. La Figure C.5(b) illustre CDL, où les blocs ont tous la même taille qui est souvent grande. Par conséquent, l'extraction des blocs ne s'effectue pas d'une façon périodique comme dans le cas de CTL, ce qui permet à CDL d'optimiser le nombre d'accès aux disques du serveur vidéo et améliore le pourcentage d'utilisation des disques. Cependant, pour pouvoir extraire plus de données d'avance et réduire le nombre d'accès aux disques, CDL requiert un buffer assez élevé.



(a) CTL



(b) CDL

Figure C.5: Techniques d'extraction pour VBR.

Nous nous limiterons dans le reste de ce chapitre à l'option de codage CBR où les blocs sont de la même taille et où durant chaque unité de service, chaque flux consomme un bloc.

Cette limitation facilite la conception du serveur vidéo, notamment pour définir la technique d'ordonnancement ou encore pour déterminer la technique de contrôle d'admission surtout dans le cas d'une architecture distribuée constituée de plusieurs noeuds et de nombreux disques.

C.2.4 Ordonnancement des flux vidéo (Stream Scheduling)

Afin de bien exploiter ses ressources, un serveur vidéo délivre les données vidéo avec un **débit de transmission** (*transmission rate*) qui est égal au **débit de consommation** (*playback rate*) chez le client. Quand on combine cette hypothèse avec la technique d'extraction périodique que nous avons adoptée, nous constatons que chaque flux est servi une fois durant chaque unité de service. Plus précisément, la quantité de données que lit un flux du serveur vidéo pendant l'unité de service i est consommée chez le client durant l'unité de service $i+1$. Étant donné que le **débit de transfert** (*transfer rate*) au niveau d'un disque est beaucoup plus élevé que le débit de consommation d'un flux, un seul disque est capable de servir plusieurs flux simultanément. La technique d'ordonnancement des flux au sein d'un disque durant la même unité de service peut avoir un impact sur le pourcentage d'utilisation efficace du disque. En effet, la technique idéale est celle qui réduit le plus le délai de positionnement sur la bonne piste et le délai de rotation afin de passer le plus de temps possible en effectuant des lectures des données. Nous considérerons principalement trois algorithmes d'ordonnancement des flux [Gemmell 95]:

1. **Round Robin (RR)**: L'ordre de chaque flux est conservé durant les différentes unités de service. Par conséquent, le temps qui sépare deux services successifs d'un flux est exactement égal à la durée d'une unité de service. La demande en buffer pour chaque flux est ainsi équivalente à la taille d'un bloc. Ce buffer est rempli durant une unité de service et aura exactement la durée d'une unité de service complète pour être vidé avant que le prochain bloc n'arrive comme l'indique la Figure C.6(a). Vu que l'ordre de lecture des différents flux est préservé, RR ne peut pas prendre en compte l'emplacement physique de blocs multiples à lire pour réduire les mouvements de la tête du disque. Par conséquent, la tête du disque risque d'effectuer des déplacements inutiles pour localiser les blocs à lire. Ceci implique des délais élevés de positionnement sur la bonne piste et, par suite une mauvaise utilisation du disque, ce qui aboutit bien évidemment à un débit de transfert faible au niveau du disque.

2. **SCAN**: Afin d'améliorer l'utilisation d'un disque et ainsi sa performance, SCAN est proposé comme technique d'ordonnancement. Contrairement à RR, SCAN ne conserve pas l'ordre suivant lequel les différents flux sont servis durant des unités de service différentes. Ceci permet de reclassifier les flux multiples durant chaque unité de service. Ainsi, la tête du disque effectue au maximum un mouvement de l'extérieur de la surface du disque vers l'intérieur et un mouvement dans la direction inverse. L'ordre des

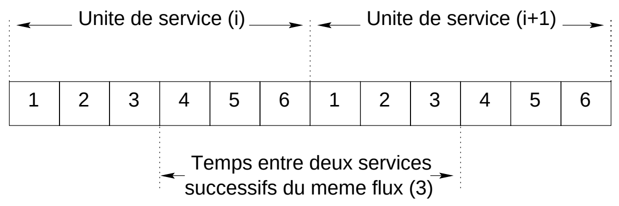
blocs à lire dépend donc de leur emplacement par rapport au chemin parcouru par la tête du disque. SCAN permet de contrôler les délais de positionnement sur la bonne piste: la somme de ces délais ne dépasse pas deux fois le délai maximum de positionnement nécessaire pour bouger la tête du disque de la piste extérieure à la piste intérieure. En revanche, le temps maximum qui sépare deux services successifs du même flux peut atteindre deux fois la durée de l'unité de service comme l'indique la Figure C.6(b). Ceci est le cas quand un flux est servi en premier lieu durant l'unité de service (i) et en dernier lieu durant l'unité de service $(i+1)$. Afin d'éviter la famine du côté du client, cette situation impose à chaque flux la réservation d'un espace de buffer équivalent à la quantité d'information qui satisfait la consommation durant deux unités de service. Rappelons que RR a besoin que de réserver un seul espace buffer; équivalent à la consommation durant une seule unité de service.

3. **GSS**: Nous avons vu que RR optimise la demande en terme de buffer mais aboutit à une mauvaise utilisation du disque et que SCAN optimise l'utilisation du disque au prix d'une demande élevée en buffer. GSS [YU 93] est proposé pour réunir l'avantage de SCAN en matière de bonne utilisation du disque et celui de RR en matière du faible besoin en buffer. Pour cela, l'unité de service est divisée en groupes et chaque flux est associé à un groupe. Entre les groupes, RR est appliqué, ce qui implique que l'ordre de servir les différents groupes soit fixe. Cependant, SCAN est utilisé au sein de chaque groupe, ce qui signifie que l'ordre de servir les flux appartenant au même groupe est variable. GSS peut donc être considéré comme un compromis entre l'optimisation du temps de recherche pour atteindre un haut débit de transfert et la réduction du besoin en buffer. Notez que GSS intègre RR et SCAN comme cas extrêmes. GSS avec un groupe est équivalent à SCAN et GSS avec autant de groupes que de flux est équivalent à RR. La Figure C.6(c) montre un exemple de GSS avec deux groupes, chacun contenant trois flux.

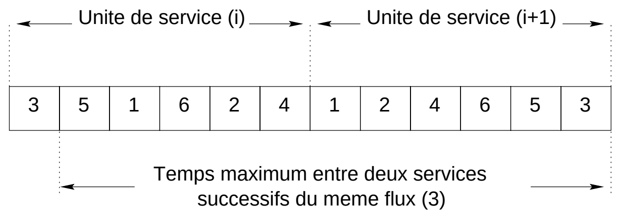
Dans le reste de ce chapitre, nous allons considérer **SCAN** comme technique d'ordonnancement des flux vidéo.

C.2.5 Contrôle d'admission (*Admission Control*)

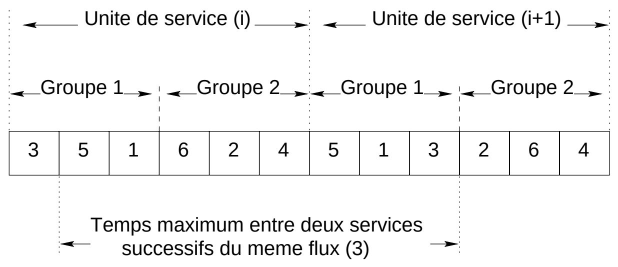
La politique de contrôle d'admission décide si un nouveau flux peut être admis. Les techniques de contrôle d'admission peuvent être déterministes ou statistiques [Gemmell 95]. Nous considérerons le cas déterministe pour le contrôle d'admission. Le nombre de flux qui peuvent être servis par un disque magnétique durant une unité de service est limité par le débit de transfert et par le temps d'accès à ce disque (délai de positionnement sur la bonne piste et délai de rotation). Soit Q_d le nombre maximum de flux que peut admettre un disque simultanément. Admettons que toutes les vidéos stockées aient le même taux de consommation r_p et soient découpées en



(a) L'algorithme d'ordonnancement RR.



(b) L'algorithme d'ordonnancement SCAN.



(c) L'algorithme d'ordonnancement GSS.

Figure C.6: Algorithmes d'ordonnancement périodique.

blocs de la même taille b. La valeur maximale d'une unité de service ne doit donc pas dépasser $\frac{b}{r_p}$ pour assurer une livraison des flux exempts de gigue (*jitter free*). Soient t_{seek} le temps maximum de positionnement sur la bonne piste et t_{rot} le délai maximum de rotation. Étant donné que l'algorithme SCAN est utilisé, le double du délai de positionnement maximal est nécessaire dans le pire des cas, pour effectuer toutes les lectures des flux pendant une unité de service $(2 \cdot t_{seek})$. Cependant, chaque flux a besoin, dans le pire des cas, d'un délai maximum de rotation (t_{rot}). Par conséquent, Q_d doit satisfaire la condition suivante: $Q_d \cdot (\frac{b}{r_d} + t_{rot}) + 2 \cdot t_{seek} \leq \frac{b}{r_p}$. Si nous considérons la valeur minimale que peut prendre l'unité de service qui est $\frac{b}{r_p}$, Q_d sera alors déterminé par:

$$Q_d = \frac{\frac{b}{r_p} - 2 \cdot t_{seek}}{\frac{b}{r_d} + t_{rot}} \qquad \text{(C.1)}$$

Pour un serveur vidéo composé de plusieurs noeuds contenant chacun des disques multiples, le contrôle d'admission devient dépendant de la technique de répartition des données à travers ses différentes composantes comme nous allons le découvrir dans la section suivante.

C.2.6 Répartition des données (Data Striping)

La méthode de **répartition des données** (*data striping*) sur les différents disques et noeuds du serveur vidéo est décisive en matière de performance et de répartition de la charge du serveur vidéo. Plusieurs travaux de recherche ont étudié la répartition de la vidéo pour les serveurs vidéo.

Afin de mieux analyser et comparer les différents algorithmes de répartition proposés, nous avons défini deux classes de répartition. La première détermine la granularité de répartition d'une vidéo entière (*Vidéo Object Striping*) tandis que la seconde détermine la granularité de répartition de chaque bloc d'une vidéo (*Video Segment Striping*). La méthode de répartition d'une vidéo détermine le nombre de disques (noeuds) qui contiennent des données appartenant à cette vidéo. En outre, sachant que nous utilisons la technique d'ordonnancement périodique SCAN, la méthode de répartition d'un bloc détermine le nombre de disques (noeuds) qui sont sollicités durant une seule unité de service pour l'extraction de ce bloc. Nous avons distingué trois méthodes de répartition d'une vidéo et trois méthodes de répartition d'un seul bloc.

Méthodes de répartition de la vidéo :

1. La vidéo est entièrement stockée sur un seul disque (*Video Single Striping* V_{ss}) [voir Figure C.7(a)].
2. La vidéo est répartie sur *quelques* disques (noeuds) du serveur vidéo (*Video Narrow Striping* V_{ns}) [voir Figure C.7(b)].
3. La vidéo est répartie sur *tous* les disques et les noeuds du serveur vidéo (*Vidéo Wide Striping* V_{ws}) [voir Figure C.7(c)].

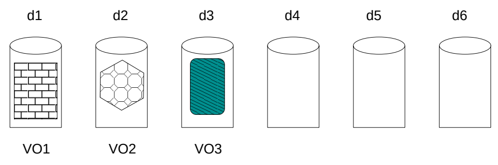
Méthodes de répartition d'un bloc :

1. Le bloc est *entièrement* stocké sur un disque (*Segment Single Striping* S_{ss}) [voir Figure C.7(d)].

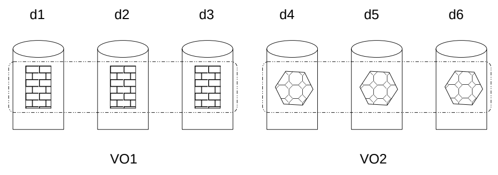
2. Le bloc est réparti sur *quelques* disques (noeuds) du serveur vidéo (*Segment Narrow Striping* S_{ns}) [[voir Figure C.7(e)].

3. Le bloc est réparti sur *tous* les disques et les noeuds du serveur vidéo (*Segment Wide Striping* S_{ws}) [[voir Figure C.7(f)].

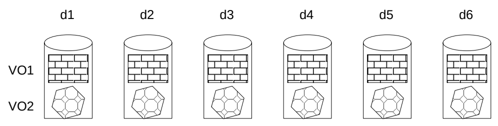
La Figure C.7 donne un exemple de chacune des six méthodes identifiées ci-dessus. Considérons un serveur vidéo contenant six disques d_i ($i \in 1, \cdots, 6$). Les termes $VO1$, $VO2$ et $VO3$ représentent trois vidéos différentes, le terme VS, un bloc.



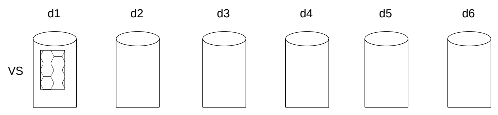
(a) Video Single Striping V_{ss}



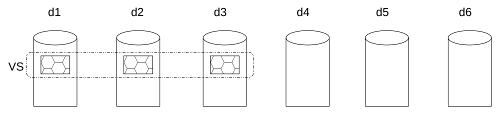
(b) Video Narrow Striping V_{ns}



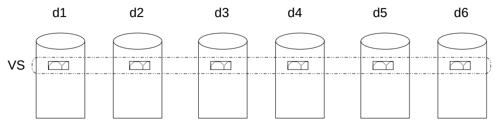
(c) Video Wide Striping V_{ws}



(d) Segment Single Striping S_{ss}



(e) Segment Narrow Striping S_{ns}



(f) Segment Wide Striping S_{ws}

Figure C.7: Méthodes de répartition de la vidéo et d'un bloc.

Un algorithme de répartition des données est identifié par la combinaison de la méthode de répartition de la vidéo avec la méthode de répartition du bloc que cet algorithme utilise. Fondé sur notre classification, le Tableau C.2 récapitule les algorithmes de répartition des données les plus significatifs. Les lignes dans ce Tableau indiquent les méthodes de répartition de la vidéo tandis que les colonnes indiquent le méthodes de répartition d'un bloc. Le signe XXX dans le Tableau indique les combinaisons irréalisables.

V_{ss} et V_{ns} n'exploitent pas la totalité de la capacité du serveur vidéo pour servir un grand nombre de clients qui demanderaient tous la même vidéo très populaire, et ils ont également des problèmes de répartition de charge. V_{ws}, en revanche, assure une parfaite répartition de la charge au

	S_{ss}	S_{ns}	S_{ws}
V_{ss}	Pas de répartition	XXX	XXX
V_{ns}	Shenoy/Vin [SHEN 97]	Shenoy/Vin [SHEN 97]	XXX
	Berson et al. [BER 94a]	Tobagi et al. [TOB 93b]	XXX
V_{ws}	Oezden et al. [OZDE 96a, OZDE 96b]	Berson et al. [BEGH 94]	Oezden et al. [OZDE 96
	Mourad [MOUR 96, Mourad 96]	Ghandeharizadeh et al.[GHAN 95b]	
	Tewari et al. [TEWA 96a]	Biersack et al. [GABI 98c]	

Table C.2: Classification des algorithmes de répartition des données

sein du serveur vidéo indépendamment des vidéos consommées et assure un haut débit pour les vidéos populaires. Par conséquent, nous allons nous limiter à V_{ws} dans nos comparaisons des méthodes de répartition des données pour un serveur vidéo. Ceci revient à ne considérer que la dernière ligne du Tableau C.2.

Comparaison des algorithmes de répartition des données (Serveur non Fiable)

Nous nous limiterons dans notre comparaison aux algorithmes de répartition des données qui sont fondés sur V_{ws} qui utilisent des méthodes différentes de répartition d'un bloc. Ces algorithmes sont l'algorithme de **répartition étroite** (*Coarse Grained Striping* **CGS**) qui combine V_{ws} avec S_{ss}, l'algorithme de **répartition large** (*Fine Grained Striping* **FGS**) qui combine V_{ws} avec S_{ws}, et finalement l'algorithme de **répartition moyenne** (*Mean Grained Striping* **MGS**) que nous avons proposé [GABI 98c] et qui combine V_{ws} avec S_{ns}. MGS divise le serveur vidéo en plusieurs groupes indépendants où chaque groupe qui contient plusieurs disques, stocke un bloc de la vidéo. Les disques d'un groupe appartiennent à des noeuds différents, rendant ainsi le serveur vidéo capable de tolérer la panne d'un noeud entier.

Fondés sur la technique d'extraction périodique, les algorithmes FGS, CGS et MGS se comportent différemment. Dans le cas de FGS qui équivaut RAID3 [Patterson 88], *tous* les disques du serveur vidéo sont impliqués durant *chaque* unité de service pour délivrer l'information vidéo à *un* client donné. Par ailleurs, pour CGS, étant équivalent à RAID5 [Patterson 88], un flux est servi par *un seul* disque durant l'unité de service i et est servi par un disque différent durant l'unité de service $i + 1$. Enfin, dans le cas de MGS, un flux est servi par *un groupe de disques* durant l'unité de service i et est servi par un groupe de disques différent durant l'unité de service $i + 1$.

Plusieurs travaux ont comparé FGS et CGS e.g. [OZDE 96a, OZDE 96b, BAAN 98]. Ces derniers ont prouvé que CGS assure un débit plus élevé que FGS. Dans [GABI 98c], nous avons comparé CGS, FGS et aussi MGS non seulement en matière du débit du serveur, mais aussi en matière du besoin en buffer, temps de latence initial pour un nouveau flux et fiabilité du serveur vidéo.

Dans le but de comparer FGS, CGS et MGS en matière du débit du serveur vidéo, nous utiliserons le critère de contrôle d'admission que nous avons introduit dans la section C.2.5 pour obtenir le nombre maximum de flux Q que peut admettre le serveur vidéo simultanément. La valeur de Q dépend surtout des performances des composantes du serveur vidéo, i.e. les valeurs de r_d, t_{seek} et t_{rot} d'un disque magnétique. La durée de l'unité de service est égale à $\tau = \frac{b}{r_p}$ comme indiquée dans Eq. C.1.

Le critère de contrôle d'admission prend la forme des Eq. C.2 (CGS), C.3 (FGS) et C.4 (MGS). Q^{CGS}, Q^{FGS} et Q^{MGS} désignent le débit total du serveur vidéo pour les trois algorithmes CGS, FGS et MGS.

Pour CGS, la totalité d'un bloc est stockée dans un disque et pour un flux, un seul bloc est lu durant une unité de service. Par conséquent, nous pouvons considérer chaque disque comme entité indépendante du reste du serveur vidéo et le débit Q du serveur vidéo est égal au débit Q_d d'un disque multiplié par le nombre total D de disques qui sont contenus dans le serveur vidéo comme l'indique Eq. C.2.

$$\frac{Q^{CGS}}{D} \cdot \left(\frac{b}{r_d} + t_{rot}\right) + 2 \cdot t_{seek} \leq \frac{b}{r_p} = \tau$$
$$\Rightarrow Q^{CGS} = \frac{\frac{b}{r_p} - 2 \cdot t_{seek}}{\frac{b}{r_d} + t_{rot}} \cdot D \tag{C.2}$$

Pour FGS, un bloc est découpé en exactement D sous-blocs et chaque sous-bloc est stocké sur un disque. Chaque disque extrait un sous-bloc de taille $\frac{b}{D}$ (voir Eq. C.3). Puisque tous les disques du serveur vidéo servent chaque flux durant chaque unité de service, FGS considère le serveur vidéo comme un disque unique, d'où la formule dans l'Eq. C.3. Noter que pour FGS, afin d'extraire un bloc de taille b pour servir un client, D accès de disques sont nécessaires contrairement au cas de CGS, où un seul accès est réalisé pour lire un bloc de la même taille b.

$$Q^{FGS} \cdot \left(\frac{\frac{b}{D}}{r_d} + t_{rot}\right) + 2 \cdot t_{seek} \leq \frac{b}{r_p}$$
$$\Rightarrow Q^{FGS} = \frac{\frac{b}{r_p} - 2 \cdot t_{seek}}{\frac{b}{D \cdot r_d} + t_{rot}} \tag{C.3}$$

MGS divise le serveur vidéo en G groupes indépendants. Chaque groupe est constitué de D_c disques. Au sein d'un groupe, la politique FGS est appliquée et, par suite, un bloc de taille b

est découpé en $\frac{b}{D_c}$ sous-blocs. Pour lire un bloc, D_c lectures sont nécessaires et par conséquent D_c accès aux différents disques du groupe sont effectués. Parmi les groupes, c'est CGS qui est appliqué, ce qui explique la valeur de Q dans Eq. C.4 qui est égale au débit d'un groupe multiplié par le nombre de groupes G.

$$\begin{aligned} \frac{Q^{MGS}}{G} \cdot \left(\frac{\frac{b}{D_c}}{r_d} + t_{rot} \right) + 2 \cdot t_{seek} &\leq \frac{b}{r_p} \\ \Rightarrow Q^{MGS} &= \frac{\frac{b}{r_p} - 2 \cdot t_{seek}}{\frac{b}{D_c \cdot r_d} + t_{rot}} \cdot G \end{aligned} \tag{C.4}$$

Notez que MGS intègre CGS et FGS comme cas particuliers: Pour $G = D$, il s'agit de CGS et pour $G = 1$, il s'agit bien de FGS.

Nos résultats ont montré que le débit du serveur vidéo avec CGS est nettement supérieur à celui du serveur avec FGS ou MGS pour la même quantité de ressources (disques, noeuds et buffer). FGS atteint le débit le plus faible. En effet, pour le même nombre de clients, le nombre d'accès aux disques que demande FGS est D fois plus élevé que celui que demande CGS et G fois plus élevé que celui demandé par MGS. Figure C.8 représente le débit du serveur vidéo pour CGS, FGS et MGS tout en considérant le même nombre de disques et la même quantité de buffer. Dans [GABI 98c], nous avons discuté en détail ce résultat et nous avons également montré qu'en matière du temps de latence initial pour un nouveau flux, MGS est le meilleur algorithme.

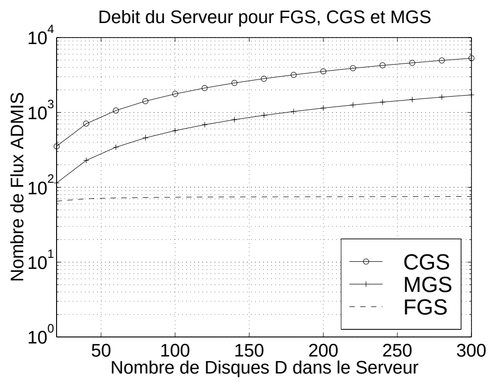
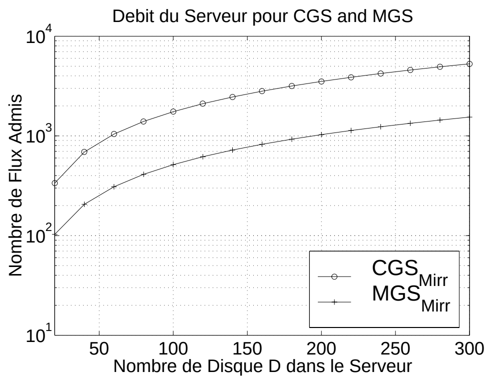


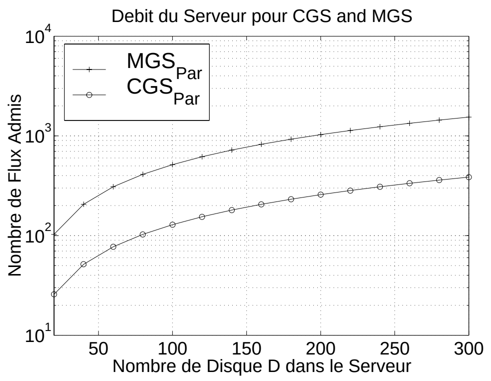
Figure C.8: Débit du serveur vidéo pour CGS, FGS et MGS ($D_c = 10$).

Comparaison des algorithmes de répartition des données (Serveur Fiable)

Dans [GABI 98c], nous avons intégré la fiabilité au sein du serveur vidéo et nous avons comparé les algorithmes CGS et MGS en matière du débit que peut atteindre le serveur vidéo. Nous avons considéré deux techniques de fiabilité: la technique de **réplication** des données (*mirroring*) et la technique de **parité** (*parity*). Après avoir étudié le besoin supplémentaire en matière de bande passante et de buffer que chacune de ces techniques de fiabilité requiert pour CGS et pour MGS, nous avons abouti à la conclusion suivante: *La technique de répartition des données et la technique de fiabilité sont* **interdépendantes**. En effet, si la réplication est utilisée, alors CGS atteint la meilleure performance en matière du débit du serveur vidéo, évidemment pour la même quantité de ressources (voir la Figure C.9(a)). Par ailleurs, si la parité est utilisée, MGS atteint un débit plus élevé que celui de CGS pour la même quantité de ressources (voir la Figure C.9(b)). Dans la Figure C.9, l'indice *Mirr* représente la réplication et l'indice *Par* la parité.



(a) Débit du serveur vidéo pour CGS et MGS avec la réplication.



(b) Débit du serveur vidéo pour CGS et MGS avec la parité.

Figure C.9: Débit d'un serveur vidéo fiable pour CGS et MGS ($D_c = 10$).

C.3 Étude de la fiabilité du serveur vidéo

Dans cette section, nous allons nous limiter à l'algorithme de répartition des données CGS. Introduire la fiabilité dans un serveur vidéo consiste principalement à y stocker l'information redondante en supplément de l'information originale. L'information redondante est utilisée en cas de panne d'une ou de plusieurs composantes du serveur vidéo afin de pouvoir régénérer l'information perdue sans que les clients s'apperçoivent d'une dégradation. Comme déjà évoqué, nous distinguons deux techniques de fiabilité d'un serveur vidéo: la réplication et la parité. Nous détaillerons tout au long de cette section les caractéristiques de chacune de ces techniques et nous les comparerons en degré de fiabilité du serveur vidéo ainsi que de performance qui est évaluée par le coût par flux.

C.3.1 Techniques et méthodes de fiabilité du serveur vidéo

Les méthodes de fiabilité utilisées dans le contexte du serveur vidéo se distinguent par la technique de fiabilité utilisée, mais aussi par la granularité de distribution de l'information redondante. Décrire une méthode de fiabilité revient donc à identifier (i) et (ii).

La Technique de réplication des données

La réplication stocke simplement une copie de chaque vidéo. Plus précisément, chaque bloc original contenu sur disque i est copié sur disque j avec $i \neq j$. Par conséquent, la réplication double le volume de stockage et double ainsi le nombre de disques nécessaire, ce qui affecte le coût du serveur vidéo. Cependant, une augmentation du nombre de disques nécessaire aboutit à une augmentation du débit total du serveur vidéo, ce qui pourrait amortir le coût supplémentaire de la réplication en admettant plus de clients. Nous examinerons en détail le coût par flux pour la réplication dans la section C.3.3.

Plusieurs chercheurs ont proposé la réplication en tant que technique de fiabilité pour un serveur vidéo, e.g. [BITT 88, MERC 95, Mourad 96, BOLO 96, CHEN 97, HSDE 90, GOMZ 92]. Nous n'allons considérer dans cet article que les mécanismes de réplication qui distribuent les blocs originaux, ainsi que leurs copies, sur tous les disques du serveur vidéo. Les copies ne sont pas donc stockées sur des disques dédiés, ce qui permet de distribuer la charge du serveur vidéo équitablement sur tous ses disques.

Pour la technique de réplication, la granularité de distribution de l'information redondante concerne deux aspects:

- Les blocs originaux d'**un** disque donné sont soit copiés sur **un** (**One-to-One**), soit **plusieurs** (**One-to-Some**), soit **tous** (**One-to-All**) les autres disques du serveur vidéo.
- Un seul bloc original est soit **entièrement** copié sur un autre disque (**Réplication entière**), soit sa copie est découpée en plusieurs **sous-blocs** qui sont stockés sur des disques différents (**Réplication en sous-blocs**).

Déterminer la technique de granularité de distribution revient donc à préciser la combinaison des deux aspects décrits ci-dessus. Comme ils dépendent de la combinaison utilisée, des critères divers peuvent être optimisés. Ces critères concernent (i) la distribution de la charge du serveur durant le mode de panne, (ii) la quantité de bande passante qu'il faut réserver sur chaque disque pour extraire l'information redondante et, par suite, le débit maximum du serveur vidéo et enfin (iii) le degré de fiabilité du serveur vidéo. Nous allons étudier plusieurs combinaisons de granularité de distribution pour la réplication en se fondant sur les trois critères (i), (ii) et (iii).

Nous distinguerons cinq méthodes différentes de réplication que nous décrirons à l'aide des exemples illustrés dans la Figure C.10. Dans ces exemples, nous considérons un serveur vidéo avec 6 disques qui stocke une vidéo composée de 30 blocs originaux.

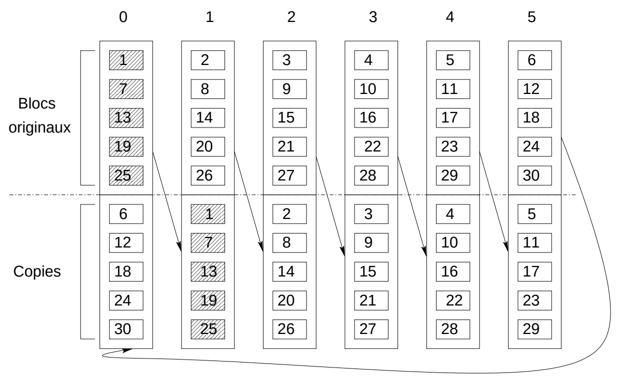
La Figure C.10(a) donne un exemple de la méthode qui consiste à répliquer les blocs originaux d'un disque sur un autre disque. Cette méthode est appelée la réplication *One-to-One* ($Mirr_{One}$). L'avantage de cette méthode est le nombre de disques qui peuvent tomber en panne sans entraîner la défaillance du serveur vidéo. Ce nombre peut atteindre, dans le meilleur des cas, $\frac{D}{2}$ si D est le nombre total des disques du serveur vidéo. En effet, dans la Figure C.10(a), même après les pannes consécutives des disques 1, 3 et 5, le serveur vidéo est toujours capable de délivrer la totalité de l'information à tous les flux. Cependant, cette méthode présente un problème de répartition de charge: dans le cas d'une panne d'un disque, les flux qui auraient dû être servis par ce dernier sont tous déplacés afin d'être servis par *un* autre disque, ce qui évoque une répartition inéquitable de la charge du serveur vidéo sur ses disques survivants.

Afin de résoudre le problème de répartition de la charge à la suite de la défaillance d'un disque du serveur vidéo la méthode *One-to-All* est utilisée comme les Figures C.10(b) et C.10(c) le montrent. La méthode illustrée dans la Figure C.10(b) utilise la réplication entière des blocs originaux ($Mirr_{all-entire}$), tandis que celle présentée dans la Figure C.10(c) utilise la réplication en sous-blocs ($Mirr_{all-sub}$). Ces deux méthodes ne peuvent tolérer que la panne d'un seul disque, ce qui peut être insuffisant pour un serveur vidéo composée de plusieurs centaines de disques.

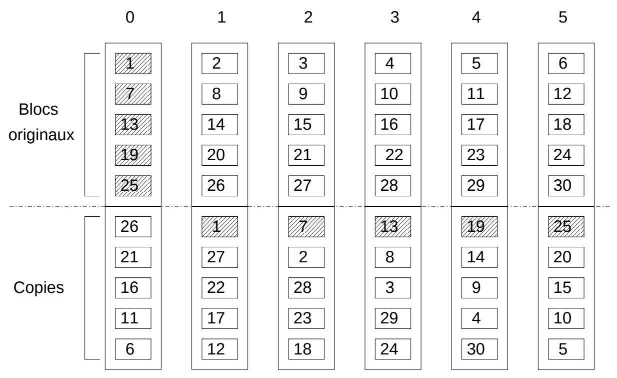
La méthode *One-to-Some* divise le serveur vidéo en plusieurs groupes indépendants. La Figure C.10(d) donne un exemple de la méthode de réplication entière des blocs originaux ($Mirr_{some-entire}$) et la Figure C.10(e) donne un exemple de la méthode de la réplication en sous-blocs ($Mirr_{some-sub}$). Pour $Mirr_{some-entire}$ et $Mirr_{some-sub}$, les groupes sont indépendants et la charge d'un disque en panne est distribuée sur les disques survivants de son groupe. Il est entendu que chaque groupe est capable de survivre à la panne d'un disque. Par conséquent, la méthode *One-to-Some*, sous ses deux formes ($Mirr_{some-entire}$ et $Mirr_{some-sub}$), assure un compromis entre (i) la distribution de la charge au sein du serveur vidéo pendant le mode de défaillance et (ii) le nombre de disques qui peuvent tomber en panne sans entraîner la défaillance du serveur vidéo.

La technique de parité

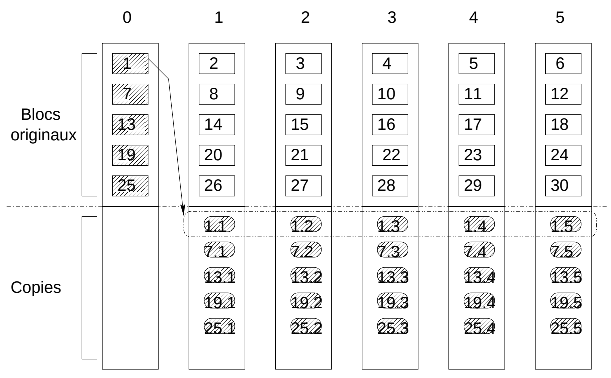
La parité consiste à générer à partir des blocs originaux des blocs de redondance que nous appelons **blocs de parité** et qui forment ensemble des groupes appelés **groupes de parités**. Un bloc de parité est créé en effectuant sur les blocs orignaux d'un groupe une simple opération **XOR** comme l'indique la Figure C.11, où un bloc P est généré à partir des blocs originaux 1, 2, 3 et 4. Un groupe de parité contient typiquement $D_c - 1$ blocs originaux et un bloc de



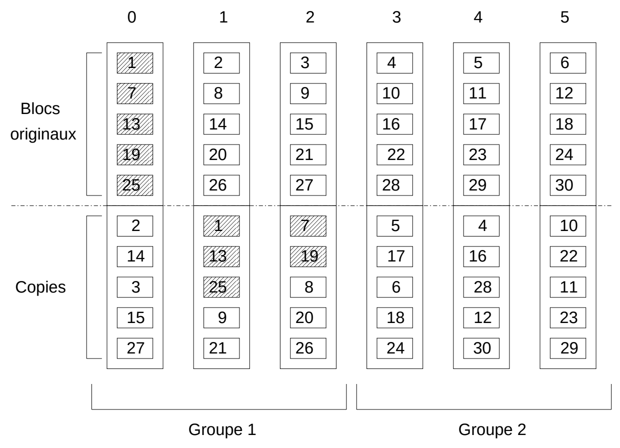
(a) *One-to-One*: $Mirr_{One}$.



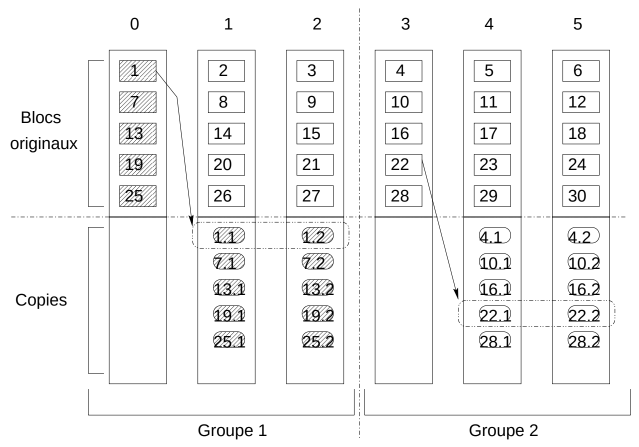
(b) *One-to-All* avec la Réplication entière: $Mirr_{all-entire}$.



(c) *One-to-All* avec la Réplication en sous-blocs: $Mirr_{all-sub}$.



(d) *One-to-Some* avec la Réplication entière: $Mirr_{some-entire}$.



(e) *One-to-Some* avec la Réplication en sous-blocs: $Mirr_{some-sub}$.

Figure C.10: Méthodes de réplication des données pour un serveur vidéo.

parité où D_c désigne la taille de ce dernier. Dans le cas d'une panne d'un disque, les blocs originaux stockés sur ce disque sont perdus. La Figure C.11 donne un exemple où le bloc 1 qui est perdu est reconstruit à la suite d'une opération XOR effectuée sur les blocs survivants (les blocs originaux 2, 3 et 4 et le bloc de parité P).

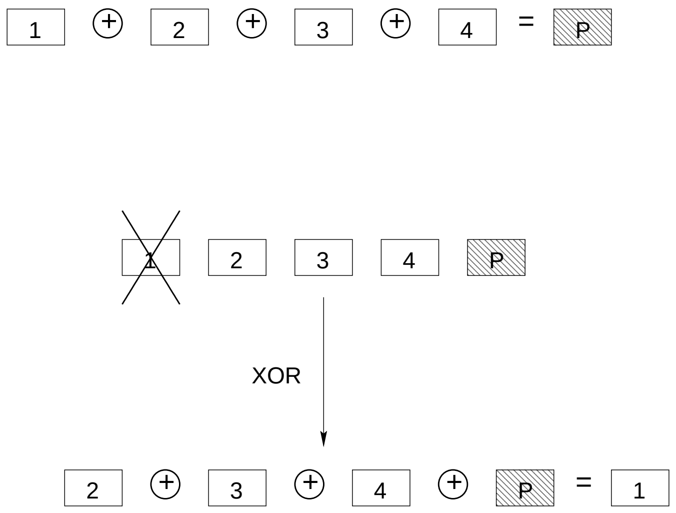
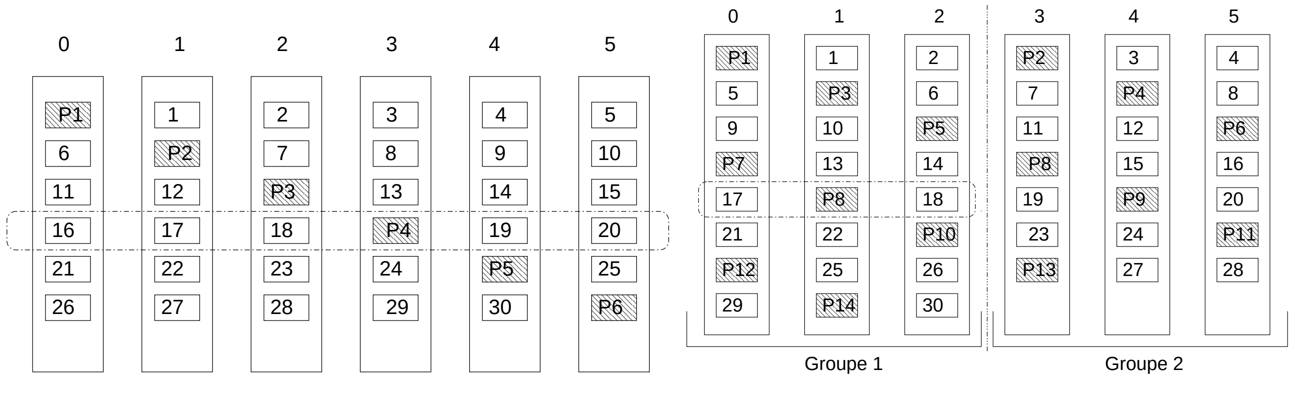


Figure C.11: Exemple de la technique de parité.

Un grand nombre de travaux ont examiné la technique de parité dans le contexte général de RAID (RAID2-6) et plus précisément dans le contexte des serveurs vidéo [LEE 92, TOB 93b, CLGK 94, HOLL 94, GHAN 95a, COHE 96, OZDE 96a, TEWA 96b, BIRK 97]. Ces travaux proposent d'assurer la fiabilité du serveur vidéo en utilisant différentes méthodes de répartition des données, e.g. RAID3 vs. RAID5, ou différents mécanismes de stockage de la parité dans le serveur vidéo. Par conséquent, ces travaux n'avaient pas forcément les mêmes objectifs. Certains essaient d'optimiser le débit du serveur, d'autres le coût du serveur ou le temps de réponse du serveur à de nouvelles requêtes ou bien encore la distribution de la charge sur les composantes du serveur vidéo. Comme dans le cas de la réplication, nous n'allons considérer que les mécanismes de parité qui distribuent les blocs de parité à travers tous les disques du serveur vidéo.

Dans le contexte de la parité, la granularité de distribution de l'information redondante se détermine par la taille d'un groupe de parité. Admettons que tous les groupes de parité soient de la même taille et que chaque disque appartienne à un seul groupe. Nous montrons dans la Figure C.12 deux exemples de méthodes de parité correspondant à RAID5. La première méthode, fondée sur le principe de *One-to-All* et appelée Par_{all}, considère que la taille d'un groupe est D, ce qui revient à prévoir un seul groupe au sein du serveur vidéo (voir La Figure C.12(a)). Un serveur vidéo qui utilise cette méthode peut survivre seulement à la panne d'un seul disque. La deuxième méthode, fondée sur *One-to-Some* et appelée Par_{some}, divise le serveur vidéo en plusieurs groupes indépendants. La taille de chaque groupe est D_c et chaque groupe peut tolérer la panne d'un de ses disques (voir la Figure C.12(b)).



(a) One-to-All: Par_{all}. (b) One-to-Some: Par_{some}.

Figure C.12: Méthodes de parité pour un serveur vidéo.

C.3.2 Modélisation de la fiabilité du serveur vidéo

Nous avons distingué cinq mécanismes de réplication des données et deux mécanismes de parité qui peuvent être utilisés pour rendre le serveur vidéo insensible aux pannes. Selon la granularité de distribution de la redondance (réplication ou parité), nous avons identifié trois classes différentes qui sont *One-to-One*, *One-to-All* et *One-to-Some*. Ces classes tolèrent des nombres différents de pannes, comme nous l'avons vu dans la section précédente. Afin de comparer *quantitativement* ces différentes classes, nous avons effectué des travaux de modélisation de la fiabilité du serveur vidéo [GAFS 99b, GABI 99b]. Nous nous sommes fondés sur des chaînes markoviennes à temps continu (*Continuous Time Markov Chains* **CTMC**) [SATR 96, HOYL 94]. Nous supposerons que la **durée moyenne de fonctionnement** de chaque disque (*disk's Mean Time To Failure* $MTTF_d$) est une fonction exponentielle du temps. Ceci est aussi valable pour la **durée moyenne de réparation** d'un disque (*disk's Mean Time To Repair* $MTTR_d$). Nous noterons respectivement λ_d et μ_d le **taux de défaillance** d'un disque (*disk's failure rate*) et le **taux de réparation** d'un disque (*disk's repair rate*): $\lambda_d = \frac{1}{MTTF_d}$ et $\mu_d = \frac{1}{MTTR_d}$.

Afin de créer le diagramme d'espace d'état (*state-space diagram*) d'un CTMC, nous faisons appel aux paramètres suivants: s désigne le nombre maximum d'états du diagramme. Un état i prend donc des valeurs entre 0 et $(s-1)$, où 0 signifie que toutes les composantes du serveur vidéo sont intactes et où $(s-1)$ est l'état de défaillance du serveur vidéo. Le paramètre $p_i(t)$ détermine la probabilité suivante: le serveur vidéo se trouve dans l'état i au temps t. Nous admettons qu'au temps $t=0$, le serveur vidéo se trouve dans l'état 0. Lorsque le serveur vidéo atteint l'état $(s-1)$, il est supposé y séjourner infiniment, ce qui revient à admettre que l'état $(s-1)$ est un **état absorbant** (*absorbing state*). Les égalités suivantes sont vérifiées: $p_0(0) = 1$, $p_i(0) = 0\ \forall i \in [1 \cdots (s-1)]$ et $p_{(s-1)}(\infty) = 1$. La fiabilité du serveur vidéo $R^s(t)$ peut être donc calculée: $R^s(t) = \sum_{i=0}^{s-2} p_i(t) = 1 - p_{(s-1)}(t)$. Nous nous limiterons dans ce résumé au cas des pannes indépendantes des disques (voir [GAFS 99b, GABI 99b] pour une étude plus détaillée).

Modélisation de *One-to-All*

Avec la méthode *One-to-All*, le serveur vidéo tombe en panne dès que deux de ses disques sont en panne. Le diagramme qui correspond à cette méthode est illustré dans la Figure C.13, où les états 0, 1 et F désignent respectivement l'état initial, l'état d'une panne d'un disque et finalement l'état de défaillance du serveur vidéo.

$$\begin{tikzcd} 0 \arrow[r, bend left, "D \cdot \lambda_d"] & 1 \arrow[l, bend left, "\mu_d"] \arrow[r, bend left, "(D-1) \cdot \lambda_d"] & F \end{tikzcd}$$

Figure C.13: Diagramme pour *One-to-All*.

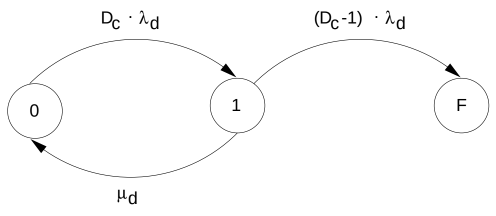
La **matrice génératrice** (*generator matrix*) Q_s du diagramme de la Figure C.13 est par conséquent:

$$Q_s = \begin{pmatrix} -D \cdot \lambda_d & D \cdot \lambda_d & 0 \\ \mu_d & -\mu_d - (D-1) \cdot \lambda_d & (D-1) \cdot \lambda_d \\ 0 & 0 & 0 \end{pmatrix}$$

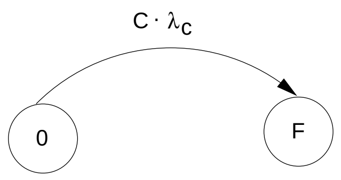
Modélisation de *One-to-Some*

La méthode *One-to-Some* divise le serveur vidéo en C groupes indépendants. Le serveur est en défaillance quand un de ses groupes tombe en panne. Nous modélisons la fiabilité d'un seul groupe (voir la Figure C.14(a)) et déduisons par la suite la fiabilité du serveur vidéo en utilisant la Figure C.14(b). Les paramètres D_c et λ_c de la Figure 14(b) désignent respectivement la taille d'un groupe et le taux de défaillance d'un groupe.

Les matrices génératrices des diagrammes de la Figure 14(b) sont Q_c (pour la Figure C.14(a)) et Q_s (pour la Figure C.14(b)):



(a) Diagramme pour un groupe.



(b) Diagramme pour le serveur.

Figure C.14: Diagrammes pour *One-to-Some*.

$$Q_c = \begin{pmatrix} -D_c \cdot \lambda_d & D_c \cdot \lambda_d & 0 \\ \mu_d & -\mu_d - (D_c - 1) \cdot \lambda_d & (D_c - 1) \cdot \lambda_d \\ 0 & 0 & 0 \end{pmatrix}$$

$$Q_s = \begin{pmatrix} -C \cdot \lambda_c & C \cdot \lambda_c \\ 0 & 0 \end{pmatrix}$$

Modélisation de ***One-to-One***

La méthode *One-to-One* est seulement significative dans le cas la réplication des données. En effet, pour la parité, cette méthode signifierait que la taille de chaque groupe de parité est égale à deux, ce qui revient à copier chaque bloc original, donc à utiliser la réplication. Comme nous avons dans l'exemple de la Figure C.10(a), plusieurs disques sont autorisés de tomber en panne sans entraîner la panne du serveur vidéo. Cependant, ceci dépend de l'emplacement des disques en panne. Si nous considérons de nouveau la Figure 10(a) et nous admettons que les deux premiers disques (0 et 1) tombent en panne successivement, le serveur vidéo se trouve dans l'état de défaillance. En d'autres termes, le serveur vidéo tombe en panne dès que deux disques consécutifs (voisins) tombent en pannes. Le nombre de pannes que peut tolérer le serveur vidéo peut donc varier entre 1 et $\frac{D}{2}$. Ce nombre ne peut pas être déterminé en avance, ce qui rend la modélisation de cette méthode assez complexe. Admettons que le serveur se trouve dans l'état $(k-1)$ et il continue à opérer. Ceci signifie que $(k-1)$ disques sont déjà tombés en pannes et tous ces disques ne sont pas voisins. Avec la panne du k^{me} disque, deux cas de figure sont envisageables: (i) le serveur continue d'opérer ou (ii) le serveur tombe en panne. Soit $P^{(k)}$ la probabilité que le cas (i) ait lieu [1]. Nous présentons dans la Figure C.15 le diagramme correspondant à la méthode *One-to-One*. Les paramètres de la Figure C.15 ont

[1][GAFS 99b] décrit en détail comment calculer $P^{(k)}$ pour chaque valeur de k

les valeurs suivantes: $\lambda_1 = D \cdot \lambda_d$, $\lambda_2 = (D-1) \cdot \lambda_d \cdot P^{(2)}$, $\lambda_3 = (D-2) \cdot \lambda_d \cdot P^{(3)}$, $\lambda_{n+1} = (D-n) \cdot \lambda_d \cdot P^{(n+1)}$, $A = (D-1) \cdot \lambda_d \cdot (1-P^{(2)})$, $B = (D-2) \cdot \lambda_d \cdot (1-P^{(3)})$, $G = (D-n) \cdot \lambda_d \cdot (1-P^{(n+1)})$, $H = (D-(n+1)) \cdot \lambda_d \cdot (1-P^{(n+2)})$, $Z = \frac{D}{2} \cdot \lambda_d$ et $\mu = \mu_d$.

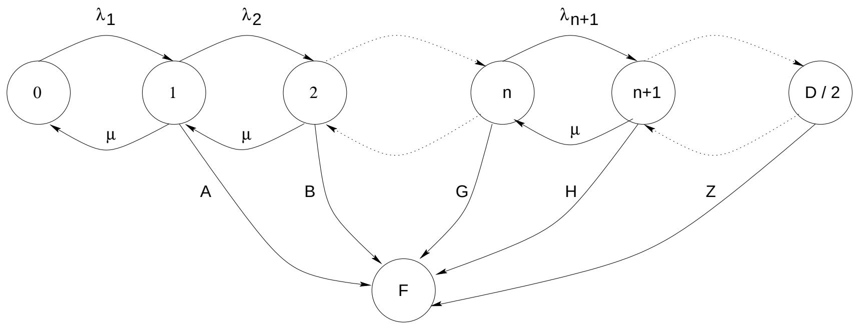


Figure C.15: Diagramme pour *One-to-One*.

La matrice génératrice Q_s de ce diagramme est par conséquent:

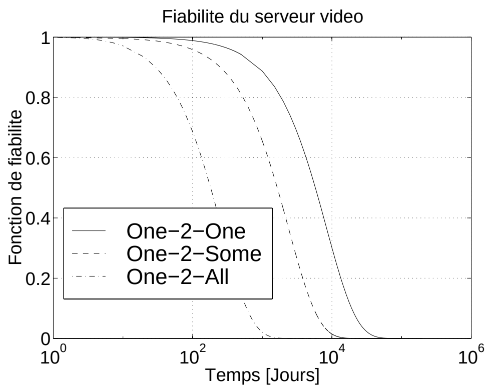
$$Q_s = \begin{pmatrix} -\lambda_1 & \lambda_1 & 0 & 0 & \cdots & 0 & 0 \\ \mu & -\mu-\lambda_2-A & \lambda_2 & 0 & \cdots & 0 & A \\ 0 & \mu & -\mu-\lambda_3-B & \lambda_3 & \cdots & 0 & B \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \mu & -\mu-Z & Z \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

Fiabilité du serveur vidéo: Résultats

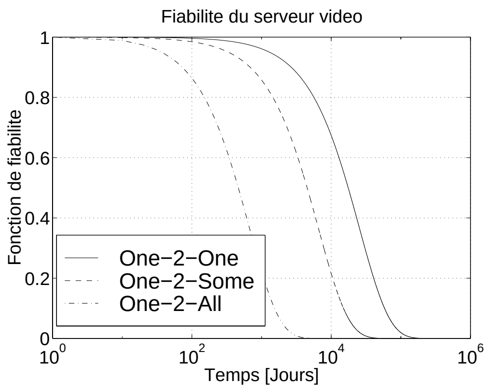
Nous résolvons nos chaînes Markoviennes (CTMC) avec l'outil **SHARPE** (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [SATR 96]. SHARPE utilise la matrice génératrice de chaque modèle pour calculer la fiabilité du serveur vidéo à un instant t donné. Les résultats en matière de fiabilité du serveur vidéo sont illustrés dans la Figure C.16. Le nombre total de disques est $D = 100$. Examinons la fiabilité du serveur pour deux valeurs de $\lambda_d = \frac{1}{60000}$ heures (la Figure C.16(a)) et $\lambda_d = \frac{1}{100000}$ heures (la Figure C.16(b)).

Les résultats montrent que, comme prévu, la fiabilité du serveur vidéo augmente quand la valeur de λ_d diminue. A titre d'exemple, après 10000 jours d'opération, la fiabilité du serveur vidéo qui utilise la méthode *One-to-One* est de l'ordre de $0,3$ pour la valeur $\lambda = \frac{1}{60000}$ heures, tandis qu'elle est de l'ordre de $0,66$ pour la valeur $\lambda_d = \frac{1}{100000}$ heures. D'autre part, la méthode *One-to-One* est la plus élevée pour les deux valeurs de λ_d. La méthode *One-to-All* obtient la fiabilité la plus basse.

Dans [GAFS 99b, GABI 99b], nous avons étudié et modélisé la fiabilité du serveur vidéo (i) dans le cas des pannes indépendantes où seuls les pannes des disques sont considérées et aussi



(a) Fiabilité du serveur vidéo pour $\lambda_d = \frac{1}{60000}$ heures.



(b) Fiabilité du serveur vidéo pour $\lambda_d = \frac{1}{100000}$ heures.

Figure C.16: Fiabilité du serveur vidéo avec $D = 100$, $\mu_d = \mu_n = \frac{1}{72}$ heures et pour *One-to-Some* $D_c = 10$.

(ii) dans le cas des panne dépendantes où les disques et les noeuds peuvent tomber en panne. Les résultats de ces travaux montrent que la méthode One-to-One devance les autres méthodes en matière de degré de fiabilité en dépit d'un coût par flux élevé. Par ailleurs, la méthode One-to-Some avec une petite taille de groupe assure le meilleur compromis entre une grande fiabilité et un faible coût par flux. One-to-All est dans tous les cas la méthode la moins performante en matière de fiabilité du serveur vidéo et de coût par flux. Nous invitons le lecteur intèressé à recourir à ces références pour une analyse plus complète.

C.3.3 Performance du serveur vidéo: Coût par flux

Débit du serveur vidéo

Comme nous utilisons CGS pour la répartition des données, le contrôle d'admission d'un nouveau flux est fondé sur l'Eq. C.2 et le débit d'un disque est celui indiqué dans l'Eq. C.1. Cependant, afin d'assurer une tolérance aux pannes, chaque disque doit réserver une part de sa bande passante (*I/O bandwidth*) disponible pour l'utiliser dans le mode de défaillance d'autres disques. Par conséquent, le débit utile de chaque disque devient inférieur à celui atteint dans l'Eq. C.1. Nous allons déterminer ce débit utile d'un disque pour chaque méthode de fiabilité.

Pour les méthodes de réplication des données qui utilisent la réplication entière des blocs ($Mirr_{one}$, $Mirr_{all-entire}$ et $Mirr_{some-entire}$), le débit est simplement la moitié de la valeur de Q_d. Prenons un exemple pour expliquer cela: un disque est tombé en panne au moment où il doit servir n flux. Au cas échéant, les blocs originaux que ces n flux auraient extraits du disque défaillant ont tous leurs copies sur *un autre* disque. Ce dernier doit donc servir ses propres flux mais aussi *tous* les flux du disque défaillant. Ainsi, afin d'assurer un service déterministe, tous les disques du serveur vidéo ne peuvent utiliser que la moitié de leur capacité en matière de bande passante pour ces trois méthodes de réplication. Soit Q^{mirr}_{entire} le débit d'un disque correspondant à ces méthodes ($Mirr_{one}$, $Mirr_{all-entire}$ et $Mirr_{some-entire}$). Nous avons donc: $Q^{mirr}_{entire} = \frac{Q_d}{2}$.

Pour les méthodes de réplication des données qui utilisent la réplication en sous-blocs, la situation est différente. Prenons tout d'abord la méthode $Mirr_{all-sub}$ (voir la Figure C.10(c)). Admettons que le débit d'un disque tolérant aux pannes est $Q^{mirr}_{all-sub}$. Dans le cas d'une panne d'un disque, chacun des disques survivants du serveur vidéo va extraire en maximum $Q^{mirr}_{all-sub}$ blocs originaux et $Q^{mirr}_{all-sub}$ sous-blocs répliqués durant chaque unité de service. Nous avons la formule suivante de contrôle d'admission:

$Q^{mirr}_{All-Sub} \cdot \left((\frac{b}{r_d} + t_{rot}) + (\frac{b^{all}_{sub}}{r_d} + t_{rot}) \right) + 2 \cdot t_{seek} \leq \tau$ où $\tau = \frac{b}{r_p}$ désigne la durée de l'unité de service. Le débit d'un disque est par conséquent: $Q^{mirr}_{All-Sub} = \frac{\tau - 2 \cdot t_{seek}}{\frac{b + b^{all}_{sub}}{r_d} + 2 \cdot t_{rot}}$ sachant que $b^{all}_{sub} = \frac{b}{(D-1)}$ est la taille d'un sous-bloc répliqué.

D'une manière analogique, nous retenons pour la méthode $Mirr_{some-sub}$ (voir la Figure C.10(e)) le débit suivant: $Q^{mirr}_{Some-Sub} = \frac{\tau - 2 \cdot t_{seek}}{\frac{b + b^{some}_{sub}}{r_d} + 2 \cdot t_{rot}}$ sachant que $b^{some}_{sub} = \frac{b}{(D_c - 1)}$ où D_c est la taille d'un groupe.

Nous calculons le débit d'un disque pour chacune des méthodes de parité. Pour la méthode Par_{all} (voir la Figure C.12(a)), chaque disque doit réserver $\frac{1}{D}^{\text{me}}$ de sa bande passante pour le mode de défaillance. Soit Q^{par}_{all} le débit d'un disque pour cette méthode: $Q^{par}_{all} = Q_d - \lceil \frac{Q_d}{D} \rceil$. D'une façon analogique, le débit d'un disque Q^{par}_{some} dans le cas de la méthode Par_{some} est $Q^{par}_{Some} = Q_d - \lceil \frac{Q_d}{D_c} \rceil$.

Besoin supplémentaire en buffer

La tolérance aux pannes exige aussi une réservation supplémentaire de ressources, notamment du buffer. Pour un serveur vidéo non fiable, le besoin en buffer B est $B = 2 \cdot b \cdot Q_s$ où b désigne la taille d'un bloc et Q_s le nombre de flux admis par le serveur vidéo. Les mécanismes de réplication des données n'ont pas besoin de buffer supplémentaire pendant le mode de défaillance. En effet, quand un disque tombe en panne, chaque bloc qui aurait dû être extrait de ce disque est lu par un autre disque contenant la copie de ce bloc. Par ailleurs, les mécanismes de parité voient leur besoin en buffer augmenter. En effet, pour reconstruire un bloc perdu, les différents blocs originaux et le bloc de parité qui appartiennent tous au même groupe de parité que le bloc perdu, doivent effectuer une opération XOR. Tous ces blocs, doivent être contenus *temporairement* dans le buffer pour effectuer l'opération XOR. Par conséquent, le besoin en buffer pour un flux est $D_c \cdot b$, avec D_c désigne la taille du groupe et b celle d'un bloc. Soit B_{all}^{par} le besoin en buffer pour la méthode Par_{all} et B_{some}^{par} le besoin en buffer pour la méthode Par_{some}. Nous avons donc: $B_{all}^{par} = D \cdot b \cdot Q_s$ et $B_{some}^{par} = D_c \cdot b \cdot Q_s$.

Coût par flux: Résultats

Nous dérivons pour chaque méthode de fiabilité considérée le coût par flux qui tient en compte le débit du serveur vidéo et son besoin en ressources (buffer et disques). Nous calculons d'abord le coût total du serveur vidéo $\$_{serveur}$ et dérivons ensuite le coût par flux $\$_{flux}$ comme suit: $\$_{flux} = \frac{\$_{serveur}}{Q_s}$. Le coût du serveur vidéo est: $\$_{serveur} = P_{buf} \cdot B + P_d \cdot V_{disque} \cdot D$, sachant que $P_{buf} = 13\$$ est le prix de $1 \ \mathrm{Mbyte}$ de buffer, B est le besoin en buffer, $P_d = 0,5\$$ est le prix de $1 \ \mathrm{Mbyte}$ de disque et V_{disque} représente la capacité de stockage d'un disque en Mbyte. Le coût par flux prend ainsi la forme suivante: $\$_{flux} = \frac{P_{buf} \cdot B + P_d \cdot V_{disque} \cdot D}{Q_s}$.

La Figure C.17 illustre le coût par flux des différentes méthodes de fiabilité du serveur vidéo considérées. Noter que le terme $Mirr_{entire}$ englobe les trois méthodes de réplication des données qui utilisent la réplication entière des blocs originaux ($Mirr_{one}$, $Mirr_{all-entire}$ et $Mirr_{some-entire}$). Ces trois méthodes ont le même débit et le même besoin en ressources et sont donc présentées par un seul terme: $Mirr_{entire}$.

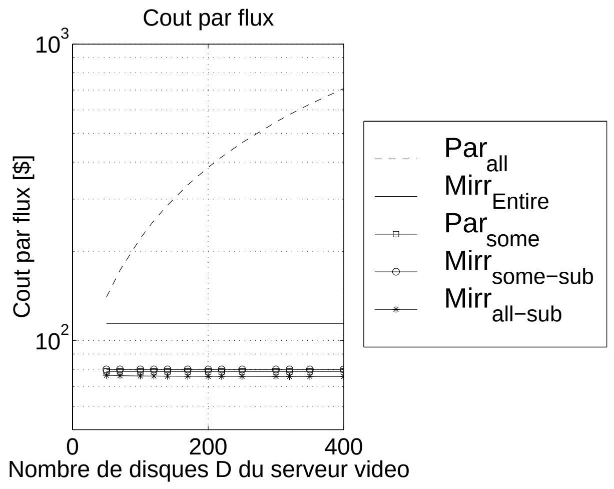


Figure C.17: Coût par flux pour les méthodes de fiabilité du serveur vidéo avec $D_c = 10$.

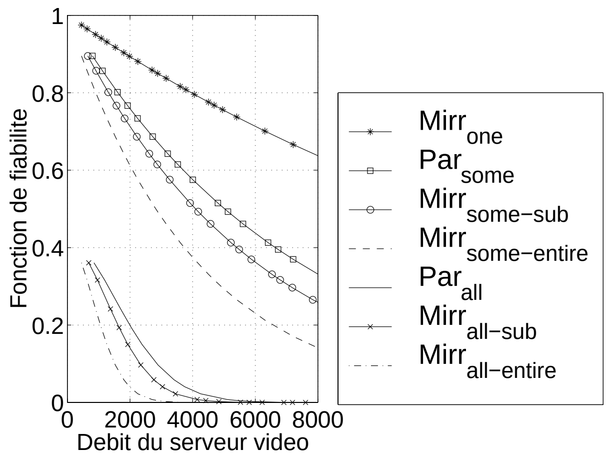
Les résultats de la Figure C.17 montrent que la méthode de parité Par_{all} est la plus chère puisqu'elle a le coût par flux le plus élevé. Pour Par_{all}, l'augmentation du coût est liée à celle du nombre total de disques D. En effet, le besoin en buffer est très grand pour Par_{all} dans le mode de défaillance et ce besoin accroît linéairement avec le nombre de disques D du serveur vidéo. Les méthodes de réplication $Mirr_{entire}$ qui utilisent la réplication entière des blocs ont un coût par flux relativement élevé. Les méthodes de réplication des données qui utilisent la réplication en sous-blocs ($Mirr_{all-sub}$, $Mirr_{some-sub}$), ainsi que la méthode de parité Par_{some}, ont des valeurs du coût par flux très proches et elles sont très basses [2].

C.3.4 Fiabilité vs. performance du serveur vidéo

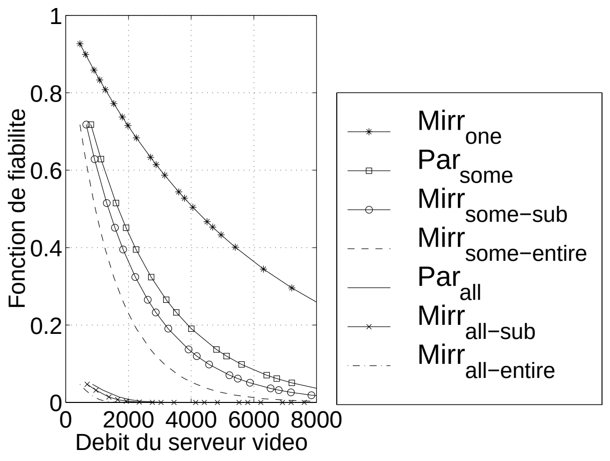
Nous avons identifié et comparé plusieurs méthodes de fiabilité du serveur vidéo. La comparaison s'est effectuée d'abord en fait du degré de fiabilité du serveur vidéo et ensuite en fait de la performance du serveur vidéo (coût par flux). Cependant, il est nécessaire de considérer à la fois le critère de fiabilité et celui de performance afin de mieux comparer ces différentes méthodes. A titre d'exemple, nous avons vu que la méthode $Mirr_{all-sub}$ a un coût par flux très bas. Néanmoins, cette méthode qui appartient à la classe *One-to-All* n'atteint qu'une fiabilité relativement faible.

La Figure C.18 illustre la fiabilité du serveur vidéo en fonction de sa performance (débit du serveur vidéo). La fiabilité du serveur vidéo est calculée après un an (Figure C.18(a)) et trois ans (Figure C.18(b)) d'opération.

Les trois mécanismes de la classe *One-to-All* (Par_{all}, $Mirr_{all-sub}$ et $Mirr_{all-entire}$) ont déjà pour des bas débits une fiabilité trop faible. Ces trois méthodes ne sont donc pas attractives pour assurer la tolérance aux pannes d'un serveur vidéo. D'après la Figure C.18(a) et la Figure C.18(b), $Mirr_{one}$ atteint la fiabilité la plus élevée.



(a) Fiabilité du serveur vidéo après 1 an d'opération.



(b) Fiabilité du serveur vidéo après 3 ans d'opération.

Figure C.18: Fiabilité du serveur vidéo pour le même débit avec $D_c = 10$, $\lambda_d = \frac{1}{100000}$ heures et $\mu_d = \mu_n = \frac{1}{72}$ heures.

[2]Voir [GAFS 99a, GAFS 99b] pour plus de détails.

En tenant compte des résultats des Figures C.17 et C.18, nous concluons que

- $Mirr_{one}$ atteint la meilleure fiabilité du serveur vidéo en dépit du coût par flux qui est relativement élevé.
- Les méthodes Par_{some} et $Mirr_{some-sub}$ ont un coût par flux très faible et une fiabilité moyenne du serveur vidéo. Notez que la taille d'un groupe de parité est $D_c = 10$ pour ces méthodes. Dans [GAFS 99b], nous avons étudié l'impact que la variation de la valeur de D_c provoque sur la fiabilité du serveur vidéo et sur le coût par flux. Nos résultats ont montré que baisser la valeur de D_c, i.e. $D_c = 3, 5$ assure un bon compromis entre une haute fiabilité du serveur vidéo et un coût bas par flux.

C.3.5 ARPS: Un Nouvel Algorithme de Placement de la Réplication

La section précédente nous a montré que la technique de réplication des données est attirante pour assurer la fiabilité d'un serveur vidéo. Cependant, pour toutes les méthodes de réplication que nous avons identifiées, le serveur vidéo doit sacrifier une partie de sa bande passante pour offrir un service ininterrompu durant le mode de défaillance. En outre, le nombre des blocs à lire de chaque disque pendant le mode de défaillance double au pire des cas, ce qui double le temps de latence (délai de positionnement sur la bonne piste et délai de rotation) nécessaire pour accéder aux différents blocs. Ceci affecte le débit de chaque disque et donc celui du serveur vidéo. Ce problème nous a guidé à conçevoir une nouvelle méthode de placement de la réplication qui évite l'augmentation du temps de latence durant le mode de défaillance et assure un débit plus haut par rapport aux méthodes classiques que nous avons discuté précédemment. Ainsi, nous avons proposé un algorithme appelé **ARPS** (*Adjacent Replica Placement Scheme*) qui stocke les sous-blocs repliqués et les blocs originaux d'une manière adjacente. Ainsi pendant le mode de défaillance, la lecture des sous-blocs repliqués se fait avec celle des blocs originaux sans qu'il y ait besoin de délai supplémentaire de positionnement sur la bonne piste ou de rotation. La Figure C.19 donne un example de ARPS où un serveur vidéo est constitué de 2 groupes, ayant chacun 3 disques.

Prenons par example le bloc 9 qui est stocké sur disque 3. La réplication de ce bloc se fait comme suit. La copie du bloc 9 est divisée en deux sous-blocs 9.1 et 9.2 qui sont stockés respectivement sur les disques 1 (adjacent au bloc 7) et 2 (adjacent au bloc 8). Dans le cas où le disque 3 tombe en panne et afin de reconstruire le bloc 9, les deux sous-blocs 9.1 et 9.2 sont lus en avance lors de la lecture des blocs 7 et 8.

D'une manière générale, ARPS utilise l'algorithme indiqué dans la Figure C.20 pour le placement de la réplication. Les paramètres utilisés dans la Figure C.20 sont définis dans le tableau C.3.

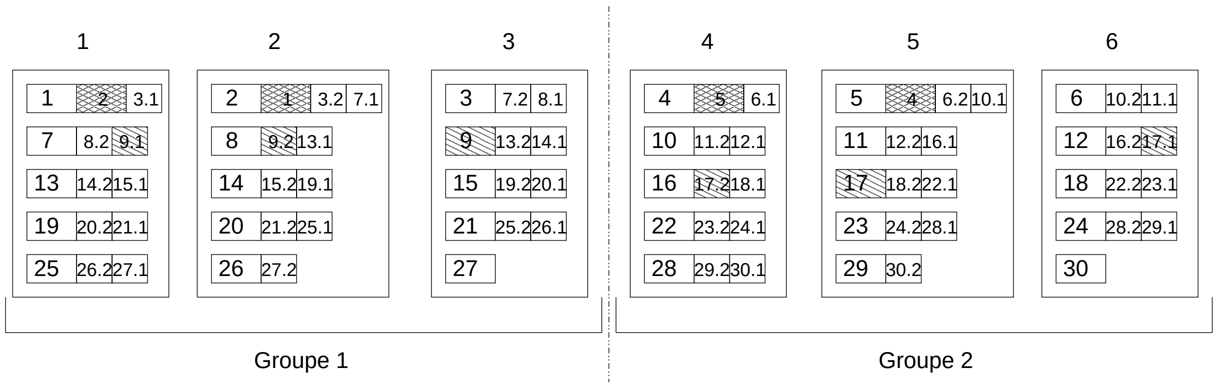


Figure C.19: Un example d'ARPS pour un serveur vidéo à 6 disques et 2 groupes.

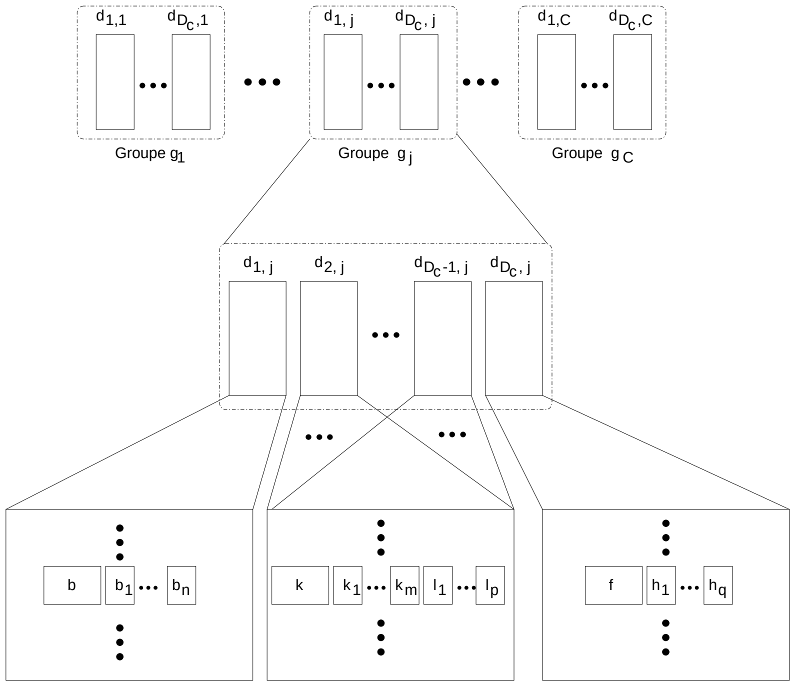


Figure C.20: Placement des données avec ARPS.

Comme la Figure C.20 l'indique, la formule de placement de la réplication dépend de la position du disque au sien d'un groupe. Nous avons identifié trois types e positions pour un groupe g_j donné.

La première position concerne le premier disque $d_{1,j}$ du groupe. Pour celui-ci, un bloc original b est stocké d'une manière adjacente avec $D_c - 1$sous-blocs répliqués. Selon Figure C.20, ces derniers sont $b_1, \cdots, b_n$ avec $b_1 = [b+1].[D_c - 1], \cdots, b_n = [b + D_c - 1].[1]$, où la notation $[\alpha].[\beta]$ identifie le β^{me} sub-bloc du bloc original α.

La deuxième position concerne tous les disques $d_{i,j}$ avec $i \in [2..D_c - 1]$. SUr ces disques,

Term	Definition
D	Le nombre total des disques
D_c	La taille d'un groupe
C	Le nombre de groupes
g_j	le j^{me} groupe du serveur, avec $j \in [1..C]$
$d_{i,j}$	Le i^{me} disque au sein du j^{me} groupe, avec $i \in [1..D_c]$ et $j \in [1..C]$

Table C.3: Data layout parameters for ARPS

le bloc original k est stocké d'une manière adjacente avec $D_c - 1$ sous-blocs répliqués qui sont $k_1, \cdots, k_m$, $l_1, \cdots, l_p$ avec $k_1 = [k+1].[D_c - 1], \cdots, k_m = [k + D_c - i].[D_c - i]$, $l_1 = [l].[D_c - (i-1)], \cdots, l_p = [l + (i-2)].[1]$ et $l = k + (D_c - i + 1) + (C-1) \cdot D_c$.

La troisième et dernière position est celle du dernier disque $d_{D_c,j}$ du groupe. Sur ce disque, le bloc original f est stocké d'une manière adjacente avec $D_c - 1$ sous-blocs répliqués qui sont $h_1, \cdots, h_q$ avec $h_1 = [h].[D_c - 1], \cdots h_q = [h + D_c - 2].[1]$, et $h = f + (C-1) \cdot D_c + 1$.

Les résultats ont montré que ARPS améliore le débit du serveur vidéo de 60 à 90 % par rapport aux algorithmes de réplication classiques tel que celui du serveur vidéo *Tiger* de Microsoft.

C.4 Implémentation du *Server Array* d'Eurecom

Nous avons parcouru les différentes étapes de la conception du serveur vidéo: d'abord, l'architecture du serveur vidéo (la matrice à serveurs) et puis les techniques d'extraction et d'ordonnancement des flux, ensuite l'algorithme de répartition des données et enfin la méthode de fiabilité. Pour chacune de ces étapes, nous avons étudié plusieurs possibilités avant de choisir la méthode la plus appropriée pour chacune de ces étapes. Dans cette section, nous décrirons briévement le prototype de serveur vidéo que nous avons implémenté a Eurecom et qui est l'objet d'un projet industriel avec France Telecom.

D'abord, le prototype de serveur vidéo intègre plusieurs de nos résultats de recherches [GAFS 98b]: L'architecture est fondée sur la matrice à serveur (voir section C.2.2). Chaque vidéo est répartie sur la totalité des noeuds et disques du *server array* (CGS). Le *server array* stocke les vidéos codés en MPEG-1 et dessert un grand nombre de clients consommant chacun des flux MPEG-1 avec $25 \;\; \text{trames}/s$ ($1,5 \;\; \text{Mbit/s}$). Il utilise des composantes hardware standards, ce qui assure son faible coût. SCAN est utilisé pour l'ordonnancement des flux et l'extraction des données est périodique. La durée de l'unité de service est déterminée par la taille des blocs à stocker. Il faut remarquer que l'ordonnancement des flux est complètement distribué et chaque noeud du *server array* sert les différents flux d'une manière autonome sans échange de messages ou synchronisation avec les autres noeuds (ordonnancement autonome). La fiabilité est assurée par la technique de réplication des données et plus précisément, le *server array* utilise la méthode $Mirr_{some-entire}$ qui divise le *server array* en plusieurs groupes indépendants (voir la section C.3). Cette méthode est capable de tolérer la panne d'un disque au sein de chaque groupe, mais aussi la panne d'un noeud entier. La détection de la panne d'un disque ou d'un noeud est dédiée au client. Ce dernier transmet le message de panne au meta serveur, qui pour sa part, déclenche la lecture des copies des blocs originaux perdus. Le client est implémenté en JAVA, ce qui lui permet d'être indépendant de la plate-forme utilisée. Il utilise en effet la classe *JMF (Java Media Framework)* qui permet la capture et la visualisation des trames vidéo codées en MPEG-1. Le client est muni des fonctionnalités interactives telles que *play, pause, reposition, stop, fast forward et fast backward*. Finalement, notez que les deux parties du *server array* – serveur et client – tournent sur des plates-formes multiples (stations UNIX et PCs). Les tests de performance effectués prouvent que le prototype du serveur vidéo est robuste au facteur de l'échelle (scalable): le débit du serveur vidéo augmente quasi linéairement quand le nombre de noeuds augmente.

C.5 Conclusions

Les applications multimédia comme la vidéo à la demande requièrent un dispositif de stockage des données audio et vidéo appelé serveur vidéo. Étant donné l'importance du volume des données vidéo, un serveur vidéo comporte habituellement de nombreux disques. Par ailleurs, la vidéo à la demande est particulièrement gourmande en capacité de mémoire et en largeur de bande passante. Ceci nécessite la conception d'un serveur vidéo avec plusieurs noeuds (machines) pour servir un grand nombre de clients. L'objectif de cette thèse est de concevoir et étudier la performance d'un tel serveur vidéo. Cette thèse identifie, propose et compare plusieurs algorithmes qui interviennent dans les différentes phases de conception d'un serveur vidéo. Elle étudie en particulier l'architecture du serveur vidéo, le placement et la distribution des données vidéo et la fiabilité du serveur vidéo. Nous proposons un algorithme de répartition des données sur plusieurs disques et noeuds du serveur vidéo, appelé Mean Grained Striping, et nous le comparons avec les algorithmes de répartition des données que nous avons identifié en matière du débit du serveur (nombre maximum des clients admis simultanément), du besoin en buffer et du temps de latence initial pour un nouveau client. Nous avons considéré le cas d'un serveur vidéo non-tolérant aux pannes et celui d'un serveur vidéo tolérant aux pannes. Nos résultats montrent surtout que l'algorithme de répartition des données et celui qui assure la fiabilité du serveur vidéo sont *interdépendants* et le choix de l'un doit être pris en combinaison avec le choix de l'autre. En outre, nous comparons plusieurs algorithmes de fiabilité du serveur vidéo en fait de la performance et du coût du serveur. Les résultats prouvent que pour un serveur vidéo, la technique de fiabilité fondée sur la simple réplication des données est moins coûteuse que celle qui est fondée sur la technique de parité. Afin d'évaluer quantitativement la fiabilité du serveur vidéo pour les différentes méthodes de fiabilité, nous modélisons la fiabilité à l'aide des chaines Markoviennes. L'évaluation de ces modèles montre que l'algorithme de fiabilité Grouped One-to-One, que nous avons proposé, assure la fiabilité la plus importante en dépit d'un coût par flux relativement élevé. Nos résultats indiquent aussi que diviser le serveur vidéo en petits groupes indépendants aboutit au meilleur compromis entre une fiabilité élevée et un coût par flux bas. Dans le cas d'un serveur vidéo qui utilise la technique de réplication des données, nous proposons une nouvelle méthode de placement de la réplication, appelée *ARPS* (Adjacent Replica Placement Scheme). Celle-ci place les données originales directement à côté des données répliquées de façon d'éliminer les temps de recherche supplémentaire quand le serveur vidéo opère dans le mode de défaillance. Nous montrons que ARPS améliore le débit du serveur vidéo de $60 - 90\%$ par rapport aux méthodes classiques de placement de la réplication. Finalement, nous implémentons un prototype de serveur vidéo qui reflète les décisions que nous avons prises durant la phase de conception. Le prototype implémente un nouvel algorithme distribué d'ordonnacement et d'extraction des données. En outre, nos résultats expérimentaux montrent que le prototype du serveur vidéo est robuste au facteur d'échelle en matière du nombre de noeuds contenus dans le serveur vidéo.

Bibliography

[BAAN 98] S. A. Barnett and G. J. Anido, "Performability of Disk-Array-Based Video Servers", *Multimedia Systems*, 6:60–74, 1998.

[BEBA 97] S. A. Barnett, G. J. Anido and P. Beadle, "Predictive call admission control for a disk array based video server", *Proceedings in Multimedia Computing and Networking*, pp. 240, 251, San Jose, California, USA, February 1997.

[BEGH 94] S. Berson, R. Muntz, S. Ghandeharizadeh and X. Ju, "Staggered Striping in Multimedia Information Systems", *Proceedings in ACM-SIGMOD Conference*, 1994.

[BER 94a] S. Berson, L. Golubchik and R. R. Muntz, "Fault Tolerant Design of Multimedia Servers", *Proceedings of SIGMOD'95*, pp. 364–375, San Jose, CA, May 1995.

[BERN 96a] C. Bernhardt and E. W. Biersack, "The Server Array: A Scalable Video Server Architecture", W. Effelsberg, A. Danthine, D. Ferarri and O. Spaniol, Eds., *High-Speed Networks for Multimedia Applications*, Kluwer Publishers, Amsterdam, The Netherlands, 1996.

[BERN 96b] C. Bernhardt and E. W. Biersack, "The Server Array: A Scalable Video Server Architecture", W. Effelsberg, O. Spaniol, A. Danthine and D. Ferrari, Eds., *High-Speed Networking for Multimedia Applications*, Kluwer Publishers, Amsterdam, The Netherlands, March 1996.

[BIRK 97] Y. Birk, "Random RAIDs with Selective Exploitation of Redundancy for High Performance Video Servers", *NOSSDAV97*, LNCS, Springer, May 1997.

[BITT 88] D. Bitton and J. Gray, "Disk Shadowing", *Proc. of the 14th int. conference on VLDB, L. A., Aug. 1988*, pp. 331–338, 1988.

[BOLO 96] W. Bolosky et al., “The Tiger Video Fileserver”, *6th Workshop on Network and Operating System Support for Digital Audio and Video,* Zushi, Japan, April 1996.

[BOLO 97] W. Bolosky, R. F. Fritzgerald and J. R. Douceur, “Distributed Schedule Management in the Tiger Video Server”, *Proc. Symp. on Operating System Principles*, pp. 212–223, October 1997.

[BUD 94] M. M. Buddhikot and G. M. Parulkar, “Design of a Large Scale Multimedia Storage Server”, *Computer Networks and ISDN Systems*, pp. 504–524, December 1994.

[BUD 95] M. M. Buddhikot, G. M. Parulkar, A. Merchant and J. R. Cox, “Distributed Data Layout, Scheduling and Playout Control in a Large Scale Multimedia Storage Server”, April 1995.

[BUDD 96] M. M. Buddhikot and G. M. Parulkar, “Efficient Data Layout, Scheduling and Playout Control in MARS”, *to apperar in ACM/Springer Multimedia Systems Journal*, 1996.

[CHA 94] E. Chang and A. Zakhor, “Admission Control and Data Placement for VBR Video Servers”, *Proceedings of the 1st International Conference on Image Processing*, pp. 278–282, Austin, Texas, November 1994.

[CHAN 93] J. Chandy and A. Reddy, “Failure evaluation of disk array organizations”, *Conf. on Distributed Computing Systems*, May 1993.

[CHAN 96] E. Chang and A. Zakhor, “Cost Analyses for VBR Video Servers”, *IEEE Multimedia*, 4(3):56–71, 1996.

[Chen 90] P. Chen and D. Patterson, “Maximizing Performance in a Striped Disk Array”, *ACM SIGARCH Conference on Computer Architecture*, Seattle, WA, May 1990.

[CHEN 93] P. Chen, E. Lee, A. Drapeau, K. Lutz, E. Miller, S. Seshan, K. Sherriff, D. Patterson and R. Katz, “Performance and Design Evaluation of the RAID-II Storage Server”, *Journal of Distributed and Parallel Databases*.

[CHEN 94a] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson, “RAID: High-Performance, Reliable Secondary Storage”, *ACM Computing Surveys*, 26(2):145–185, June 1994.

[CHEN 97] M.-S. Chen et al., “Using Rotational Mirrored Declustering for Replica Placement in a Disk-Array-Based Video Server”, *Multimedia Systems*, 5(3):371–379, December 1997.

[CHENMS 93] M.-S. Chen, D. D. Kandlur and P. S. Yu, “Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams”, *Proc. 1st ACM Conference on Multimedia*, Anaheim, CA, August 1993.

[Chervenak 94] A. L. Chervenak, *Tertiary Storage: An Evaluation of New Applications*, Ph.D. Thesis, University of California, Berkeley, 1994.

[Chung 96] S. M. Chung, Ed., *Multimedia Information Starage and Mangement*, Kluwer Academic Publishers, Boston/London/Dordrecht, 1996.

[CLGK 94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson, “RAID: High-Performance, Reliable Secondary Storage”, *ACM Computing Surveys*, 1994.

[Cliff 96] M. Cliff, N. P.S., O. Banu, R. Rajeev and S. Avi, *The Fellini Multimedia Storage Server*, chapter 5, pp. 117–146, Multimedia Information Starage and Mangement, Kluwer Academic Publishers, 1996.

[COHE 96] A. Cohen and W. Burkhard, “Segmented Information Dispersal (SID) for Efficient Reconstruction in Fault-Tolerant Video Servers”, *Proc. ACM Multimedia 1996*, pp. 277–286, Boston, MA, November 1996.

[DENG 96] J. Dengler, C. Bernhardt and E. Biersack, “Deterministic Admission Control Strategies in Video Servers with Variable Bit Rate Streams”, *Proceedings of the European Workshop on Interactive Distributed Multimedia Systems and Services*, Lecturenotes in Computer Science, Springer, March 1996.

[Doganata 96] Y. N. Doganata and A. N. Tantawi, *Storage Hierarchy in Multimedia Servers*, chapter 3, pp. 61–94, Multimedia Information Storage and Management, Kluwer Academic Publishers, 1996.

[Exner 99] J. Exner, “A Reliable Video Server Based on Mirroring: Design, Implementation, and Performance Analysis”, M.S. Thesis, University of Karlsruhe/Institut Eurecom, Sophia Antipolis, France, January 1999.

[GABI 97] J. Gafsi and E. W. Biersack, “Comparison of Shared and Dedicated Buffer Management Strategies”, , Institut Eurecom, December 1997.

[GABI 98c]	J. Gafsi and E. W. Biersack, “Data Striping and Reliablity Aspects in Distributed Video Servers”, *In Cluster Computing: Networks, Software Tools, and Applications*, 2 (1):75–91, February 1999.
[GABI 99a]	J. Gafsi and E. W. Biersack, “A Novel Replica Placement Strategy for Video Servers”, *Proceedings of the 6th International Workshop On Interactive and Distributed Multimedia Systems IDMS’99, Toulouse, France*, October 12-15 1999.
[GABI 99b]	J. Gafsi and E. W. Biersack, “Modeling and Performance Comparison of Reliability Strategies for Distributed Video Servers”, *to appear in IEEE Transactions on Parallel and Distributed Systems*, February 2000.
[GAFS 98b]	J. Gafsi, U. Walther and E. W. Biersack, “Design and Implementation of a Scalable, Reliable, and Distributed VOD-Server”, *Proceedings of to the 5th joint IFIP-TC6 and ICCC Conference on Computer Communications*, October 1998.
[GAFS 99a]	J. Gafsi and E. W. Biersack, “Performance and Cost Comparison of Mirroring- and Parity-Based Reliability Schemes for Video Servers”, *Proceedings of KiVS’99*, Darmstadt, Germany, March 1999.
[GAFS 99b]	J. Gafsi and E. W. Biersack, “Performance and Reliability Study for Distributed Video Servers: Mirroring or Parity?”, *Proceedings of the IEEE international conference on multimedia computing and systems (ICMCS’99)*, Florence, Italy, June 1999.
[Gafsi 99]	J. Gafsi and E. W. Biersack, *Serveurs Video: Architecture et Perfromance*, Applications Multimedia, Hermes Science Publications, 1999.
[GB 97a]	J. Gafsi and E. Biersack, “Impact of Buffer Sharing in Multiple Disk Video Server Architecture”, *Proceedings in the 6th Open Workshop on High Speed Networks*, Stuttgart, Germany, October 1997.
[Gemmell 95]	D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan and L. A. Rowe, “Multimedia Storage Servers: A Tutorial and Survey”, *IEEE Computer*, 28(5):40–49, May 1995.
[GHAN 95a]	S. Ghandeharizadeh and S. H. Kim, “Striping in Multi-disk Video Servers”, *Proc. High-Density Data Recording and Retrieval Technologies Conference*, SPIE, October 1995.

[GHAN 95b] S. Ghandeharizadeh and H. K. Seon, "Striping in multi-disk video servers", *Proceedings in the SPIE International Symposium on Photonics Technologies and Systems for Voice, Video, and Data Communications*, 1995.

[Ghandeharizadeh 98] S. Ghandeharizadeh, R. Zimmermann, D. Ierardi and T.-W. Li, *Mitra: A Scalable Continuous Media Server*, chapter 3, pp. 63–90, Multimedia Technologies and Applications for the 21st Century, Kluwer Academic Publishers, Furth, Borko edition, 1998.

[GIBS 90] G. A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, Ph.D. Thesis, University of California at Berkley, December 1990.

[GKSZ 96] S. Ghandeharizadeh, S. H. Kim, C. Shahabi and R. Zimmermann, *Placement of Continuous Media in Multi-Zone Disks*, chapter 2, pp. 23–59, Multimedia Information Storage and Management, Kluwer Academic Publishers, Soon M. Chung edition, 1996.

[GOLP 98] L. Golubchik, J. C.-S. Lui and M. Papadopouli, "A Survey of Approaches to Fault Tolerant Design of VOD Servers: Techniques, Analysis, and Comparison", *Parallel Computing Journal*, 24(1):123–155, 1998.

[GOMZ 92] L. Golubchik, J. C. Lui and R. R. Muntz, "Chained Declustering: Load Balancing and Robustness to Skew and Failures", *In Proceedings of the Second International Workshop on Research Issues in Data Engineering: Transaction and Query Processing, Tempe, Arizona*, pp. 88–95, Tempe, Arizona, February 1992.

[Grochowski 97a] E. Grochowski, "Disk Drive Price Decline", , IBM Almaden Research Center, San Jose, California, 1997.

[Grochowski 97b] E. Grochowski, "IBM Hard Disk Drive Evolution", , IBM Almaden Research Center, San Jose, California, 1997.

[Grochowski 97c] E. Grochowski, "Internal (Media) Data Rate Trend", , IBM Almaden Research Center, San Jose, California, 1997.

[Hennessy 90] J. L. Hennessy and D. A. Patterson, *Computer Arcitecture A Quantative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[HOLL 94] M. Holland, G. Gibson and D. Siewiorek, “Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays”, *Journal of Distributed and Parallel Databases*, 2(3), July 1994.

[HOYL 94] A. Hoyland and M. Rausand, *System Reliability Theory: Models and Statistical Methods*, volume 518, John Wiley and Sons, 1994.

[HSDE 90] H. I. Hsiao and D. J. DeWitt, “Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines.”, *In Proceedings of the Int. Conference of Data Engeneering (ICDE), 1990*, pp. 456–465, 1990.

[Kaddeche 98] H. Kaddeche, *Etude des Performances et de la Tolerance aux Pannes de Serveurs Multimedias Multidisques*, Ph.D. Thesis, Universite Pierre et Marie Curie (Paris 6), July 1998.

[KATZ 92] R. Katz, P. Chen, A. Drapeau, E. Lee, K. Lutz, E. Miller, S. Seshan and D. Patterson, “RAID-II: Design and Implementation of a Large Scale Disk Array Controller”, UCB/CSD-92-705, Computer Science Division, UCB, Berkeley, CA, October 1992.

[KIENZ 95] M. G. Kienzle, A. Dan, D. Sitaram and W. Tetzlaff, “Using Tertiary Storage in Video-on-Demand Servers”, *Proceedings of the IEEE COMPCON’95*, San Francisco, CA, March 1995.

[Korst 97] J. Korst, “Random Duplicated Assignment: An Alternative to Striping in Video Servers”, *ACM Multimedia, Seattle, USA*, 1997.

[LEE 92] E. K. Lee et al., “RAID-II: A Scalabale Storage Architecture for High-Bandwidth Network File Service”, UCB/CSD 92/672, University of California, Berkeley, February 1992.

[LEED 93] E. K. Lee, *Performance Modeling and Analysis of Disk Arrays*, Ph.D. Thesis, University of California at Berkley, 1993.

[Lu 96] G. Lu, *Communication and Computing for Distributed Multimedia Systems*, Artech House Publishers, Boston/London, 1996.

[Mancini 99] T. Mancini and B. Frison, “Conception et Implementation en Java de la Partie Client d’un Serveur Video”, , Insitut Eurecom, April 1999.

[MATR 93] M. Malhotra and K. S. Trivedi, “Reliability Analysis of Redundant Arrays of Inexpensive Disks”, *Journal of Parallel and Distributed Computing*, 17:146–151, 1993.

[MERC 95] A. Merchant and P.-S. Yu, "Analytic Modeling and Comparisons of Striping Strategies for Replicated Disk Arrays", *IEEE Transactions on Computers*, 44(3):419–433, March 1995.

[MOUR 96] A. Mourad, "Doubly-Striped Disk Mirroring: Reliable Storage for Video Servers", *Multimedia, Tools and Applications*, 2(3):253–272, May 1996.

[Mourad 96] A. Mourad, "Issues in the Design of a Storage Server for Video-On-Demand", *Multimedia Systems*, 4(2):70–86, 1996.

[Mr.X] Seagate Disc Home, http://www.seagate.com/disc/disctop.shtml.

[OZDE 96a] B. Ozden et al., "Fault-tolerant Architectures for Continuous Media Servers", *SIGMOD International Conference on Management of Data 96*, pp. 79–90, June 1996.

[OZDE 96b] B. Ozden et al., "Disk Striping in Video Server Environments", *Proc. of the IEEE Conf. on Multimedia Systems*, pp. 580–589, Hiroshima, Japan, jun 1996.

[Patterson 88] D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pp. 109–116, Chicago, IL, June 1988.

[Randolph 95] N. Randolph, *Probability, Stochastic Processes, and Queeing Theory*, Springer-Verlag, 1995.

[REDD 93] A. Reddy et al., "Design and Evaluation of Gracefully Degradable Disk Arrays", *J. of Parallel and Distributed Algorithms and Architectures*, 17:28–40, 1993.

[REDD 96] A. Reddy and R. Haskin, "Video Servers", *The Communications Handbook*, CRC Press, 1996.

[SaGa 86] K. Salem and H. Garcia-Molina, "Disk Striping", *IEEE International Conference on Data Engineering*, 1986.

[SATR 96] R. A. Sahner, K. S. Trivedi and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*, Kluwer Academic Publishers, 1996.

[Shastri 98] V. Shastri, V. P. Rangan and S. Sampath-Kumar, *DVDs: Much Needed ”Shot in the Arm” for Video Servers*, chapter 2, pp. 31–61, Multimedia Technologies and Applications for the 21st Century, Kluwer Academic Publishers, 1998.

[SHEN 97] P. Shenoy and H. Vin, “Efficient Striping Techniques for Multimedia File Servers”, G. Parulkar, Ed., *NOSSDAV 97*, May 1997.

[Srivastava 97] A. Srivastava, A. Kumar and A. Singru, “Design and Analysis of a video-on-demand server”, *ACM Multimedia Systems*, 5:238–254, 1997.

[STIEW 82] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable Systems Design*, Digital Press, Bedford, Massachusetts, 1982.

[TEWA 96a] R. Tewari, D. M. Dias, W. Kish and H. Vin, “Design and Performance Tradeoffs in Clustered Video Servers”, *Proceedings IEEE International Conference on Multimedia Computing and Systems (ICMCS’96)*, pp. 144–150, Hiroshima, June 1996.

[TEWA 96b] R. Tewari, D. M. Dias, W. Kish and H. Vin, “High Availability for Clustered Multimedia Servers”, *Proceedings of International Conference on Data Engineering*, New Orleans, LA, February 1996.

[TEWA 96c] R. Tewari, R. King, D. Kandlur and D. M. Dias, “Placement of Multimedia Blocks on Zoned Disks”, *Proceedings of IS and T/SPIE Multimedia Computing and Networking*, San Jose, January 1996.

[TOB 93b] F. A. Tobagi, J. Pang, R. Baird and M. Gang, “Streaming RAID(tm) – A Disk Array Management System For Video Files”, *Proceedings of the 1st ACM International Conference on Multimedia*, Anaheim, CA, August 1993.

[Walter 97] U. Walter, “Design and Implementation of a distributed, reliable MPEG Video Server on ATM Networks using Forward Error Correction Methods”, M.S. Thesis, University of Karlsruhe/Institut Eurecom, Sophia Antipolis, France, December 1997.

[WILK 94] C. Ruemmler and J. Wilkes, “An introduction to disk drive modeling”, *IEEE Computer*, 27(3):17–28, March 1994.

[WILK 96] J. Wilkes, R. Golding, C. Staelin and T. Sullivan, “The HP AutoRAID Hierarchical Storage System”, *ACM Transactions on Computer Systems*, 14(1), February 1996.

[WORT 95] B. L. Worthington, G. Ganger, Y. N. Patt and J. Wilkes, “On-Line Extraction of SCIS Drive Characteristics”, *Proc. 1995 ACM SIGMETRICS*, pp. 146–156, Ottawa, Canada, May 1995.

[YU 93] P. S. Yu, M.-S. Chen and D. D. Kandlur, “Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management”, *ACM Multimedia Systems*, 1(3):99–108, 1993.

List of Publications

Journal Papers

[GABI 98c] J. Gafsi and E. W. Biersack, "Data Striping and Reliablity Aspects in Distributed Video Servers", *In Cluster Computing: Networks, Software Tools, and Applications*, 2 (1):75–91, February 1999.

[GABI 99b] J. Gafsi and E. W. Biersack, "Modeling and Performance Comparison of Reliability Strategies for Distributed Video Servers", *to appear in IEEE Transactions on Parallel and Distributed Systems*, February 2000.

[Gafsi 99] J. Gafsi and E. W. Biersack, *Serveurs Video: Architecture et Perfromance*, Applications Multimedia, Hermes Science Publications, 1999.

Conference Papers

[GAFS 98b] J. Gafsi, U. Walther and E. W. Biersack, "Design and Implementation of a Scalable, Reliable, and Distributed VOD-Server", *Proceedings of to the 5th joint IFIP-TC6 and ICCC Conference on Computer Communications*, October 1998.

[GAFS 99a] J. Gafsi and E. W. Biersack, "Performance and Cost Comparison of Mirroring- and Parity-Based Reliability Schemes for Video Servers", *Proceedings of KiVS'99*, Darmstadt, Germany, March 1999.

[GAFS 99b] J. Gafsi and E. W. Biersack, "Performance and Reliability Study for Distributed Video Servers: Mirroring or Parity?", *Proceedings of the IEEE international conference on multimedia computing and systems (ICMCS'99)*, Florence, Italy, June 1999.

[GABI 99a] J. Gafsi and E. W. Biersack, "A Novel Replica Placement Strategy for Video Servers", *Proceedings of the 6th International Workshop On Interactive and Distributed Multimedia Systems IDMS'99, Toulouse, France*, October 12-15 1999.

MISC

[GB 97a] J. Gafsi and E. Biersack, “Impact of Buffer Sharing in Multiple Disk Video Server Architecture”, *Proceedings in the 6th Open Workshop on High Speed Networks*, Stuttgart, Germany, October 1997.

[GABI 97] J. Gafsi and E. W. Biersack, “Comparison of Shared and Dedicated Buffer Management Strategies”, , Institut Eurecom, December 1997.