A Study on High Throughput Large File Sharing System for a Global Environment and its Applications

August 2017

Daisuke Ando

A Thesis for the Degree of Ph.D. in Engineering

A Study on High Throughput Large File Sharing System for a Global Environment and its Applications

August 2017

Graduate School of Science and Technology
Keio University

Daisuke Ando

Acknowledgement

Firstly, I would like to express my greatest appreciation to my adviser Senior Lecturer Kunitake Kaneko for the continuous support of my Ph.D. study and related research. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I also would like to express my special appreciation to Prof. Fumio Teraoka. His guidance helped me in all the time of research. He has been a tremendous mentor for me, and his guidance has been always valuable. Without his permanent help, this thesis would no have been possible.

My sincere thanks also go to Prof. Jason Leigh for welcoming me to the LAVA in the University of Hawaii. It was a great experience for me to study in LAVA under his enthusiastic support. He also gave me worth comments and warm encouragement as my committee member.

I owe my deepest gratitude to Prof. Yamato Sato for giving me valuable opportunities and advices on research in Graduate School of Business and Commerce. He made my experience in the Graduate School of Business and Commerce invaluable.

I would like to thank the rest of my thesis committee: Prof. Kenji Kono, Prof. Hiroaki Nishi, Prof. Tomohiro Kudoh for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

I would like to offer my special thanks to RAs, mentors, and professors in the Program for Leading Graduate School for ”Science for Development of Super Mature Society” for their invaluable and warm supports. In addition, I would particular like to thank every member of Kaneko Laboratory, Teraoka Laboratory, Sato Yamato Seminar, and LAVA for encouraging me and helping me in various ways.

Finally, I would like to show my greatest appreciation to my family for their unconditional supports and encouragements.

Abstract

Large file sharing using the Internet among people and organizations around the world has now become widespread in a variety of industries, with the rapid growth of multimedia devices, networking technologies, and cloud technologies. For example, video production companies usually share source video files with post-production companies and contract out post-production processes like video editing for low cost content creation. They need to share large numbers of large files at low cost and high throughput regardless of network conditions such as network delays and packet loss rates. Although low cost and high throughput file sharing on the Internet is in high demand, it remains challenging to achieve because of the TCP-based protocols usually employed to deliver files, such as HTTP and FTP. TCP performance degrades when the network delay between a server and a client is significant and/or the packet loss rate is high. While creating caches of files using a content distribution network (CDN) improves file sharing performance, it comes with a high storage cost.

This thesis proposes Content Espresso, a large file sharing system for a global environment, and its applications. To be successful, Content Espresso must meet eight requirements: 1) High throughput file transmission regardless of network delay and packet loss, 2) high throughput storage IO, 3) low storage demands with high availability, 4) appending appropriate redundancy to each stored file, 5) low storage device preparation and management cost, 6) sharing a large number of files globally, 7) maintaining metadata and access control information securely, and 8) authenticating and authorizing the users. In order to satisfy these requirements, the fundamental mechanism of Content Espresso is designed as follows: 1) Redundancy data is appended by forward error correction (FEC) to the original file; 2) the original file and redundancy data are split into chunks and stored in globally dispersed storage servers called Chunk Servers; 3) stored chunks are delivered to the client on request using a User Datagram Protocol (UDP); and 4) non-received chunks caused by storage failure or network packet loss can be recovered by FEC in the client. Content Espresso consists of four main modules: File Manager, Storage Allocator, Chunk Generator, Cluster Head, and Chunk Server. Content Espresso is installed on 79 physical machines, including 72 Chunk Servers, and evaluated by emulating global environments using the `tc` command. The results confirm that Content Espresso achieves stable file retrieval faster than 3Gbps, regardless of network delay and packet loss rate.

This thesis also proposes two Content Espresso applications: Demitasse, a network-

oriented uncompressed UHD video playback system, and a feature that improves the file sharing performance of web-based collaboration systems. The goal of Demitasse is to retrieve stored video component files from Content Espresso and play them back at the requested frame rate. Demitasse stores the uncompressed frame files as video component files in Content Espresso and the relationship between the frame files in its catalogue system. Demitasse was designed, implemented, and evaluated using 79 physical machines, including 72 Chunk Servers. The results confirm that Demitasse has the ability to retrieve uncompressed UHD video frame files from Content Espresso and play them back at 30fps. In the second application, Content Espresso improves the file sharing performance of web-based collaboration systems, which typically operate using web browsers. In this thesis, SAGE2 is chosen to serve as an example of such a system. SAGE2 shares information on large high-resolution displays with other sites, enabling people at multiple sites to work together in front of the displays. In this application, Content Espresso-based (CE-based) file sharing is proposed for SAGE2 by introducing a relay server. The relay server receives file retrieval requests from web browsers by HTTP and relays them to Content Espresso using Content Espresso API. Then, the relay server receives the file data from Content Espresso and relays it to the requesting web browsers. This relay deals successfully with the reality that Content Espresso cannot receive file retrieval requests from web browsers directly and that web browsers cannot receive chunks delivered by UDP. The performance of the CE-based file sharing mechanism is evaluated by comparing file sharing time using the CE-based mechanism with that using the original SAGE2 mechanism. The results confirm that the CE-based file sharing in SAGE2 improves file sharing performance when the round-trip time (RTT) to the remote site is larger than 10ms. The proposed mechanism can be applied to web-based collaboration systems other than SAGE2 because it is designed and implemented using existing browser-based technologies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

File storing and sharing on the Internet by people around the world has now become widespread, with the rapid growth of multimedia devices, networking technologies, and cloud technologies. The growth of multimedia devices enables people to own high-performance smartphones that can easily create high quality multimedia content such as Full HD (1920 x 1080) and UHD (3840 x 2160) videos. A recent global survey reports that 43% of people own a smartphone [3]. High-speed mobile networking technologies such as 3G and LTE enable people to upload their own creative content files quickly to Internet-based file storing and sharing services such as Dropbox [4] and Google Drive [5]. In addition, cloud storage technologies have decreased the cost of file storing and sharing services. As a result of these technological improvements and sharing, file storing and sharing on the Internet has become widely popular, and shows no sign of slowing.

1.2 Definition of file sharing in a global environment

There are two main ways to share a file: physical sharing and network-based sharing. In the former, most fundamental form of file sharing, the file sender writes the files to physical media such as an optical disc, an HDD, or a USB memory stick and physically gives the media to the file recipient. When file senders and recipients can meet in person, physical-based file sharing is fast, easy, and inexpensive. Where a meeting is not possible, the file sender has to ship the physical media to the file recipient, can cause delays and increase cost. When the file recipient receives the physical media, the file recipient copies the files to the local storage of the computer.

In network-based sharing, by contrast, the file sender transfers the file to the file recipient over a network. Network-based sharing is not affected by the location of the sender or the recipient. Files can be shared as long as the sender's and recipient's computers are connected in some fashion. Examples of network-based sharing include sending the file

via email, transferring it through an HTTP or FTP server, giving the file's URL to the file recipients, and using a cloud storage service. The file sender usually tries to select the best way to send the file by taking file sharing time and cost into account. The appropriate method varies by file size, location, and number of file recipients. For example, when a sender shares a small file with a single recipient and the sender and recipient can meet in person, the sender would store the file on a USB memory stick and give it to the recipient physically. When the recipient is located at some distance from the sender, the file sender might use e-mail to share the file. Table 1.1 displays the typically appropriate sharing method for each situation.

This thesis focuses on large file sharing in a global environment, which is defined in the thesis as follows; the file sender and the file recipient are connected by the Internet, the RTT between the file sender and the file recipient is typically long, and there is more than one recipient. The target environment of this thesis thus involves large file size, a wide area network (WAN) rather than a local area network (LAN), and multiple recipients, as shown in Table 1.1.

There are two main approaches two network-based file sharing in the global environment; the client-server file sharing approach and the P2P file sharing approach, which are described in Figure 1.1. In the client-server approach, the sender uploads the file to the storage server and the recipient retrieves the file from that same storage server. Although this approach demands two steps – uploading to the storage server and retrieval from the storage server – it provides stable availability regardless of network condition or hardware of either the sender or the recipient. With the P2P file sharing approach, by contrast, the file sender communicates with the file recipient directly. Both sender and recipient have to know information such as an IP address before beginning to share files. Since P2P availability does depend on the networks and hardware used by sender and recipient, it is difficult to draw general conclusions as to how easily it can be used. Therefore, this thesis adopts the client-server file sharing approach.

1.3 Difficulties of network-based file sharing on the Internet

The rapid growth of networking technologies has enabled the use of network-based file sharing over the Internet. This section presents the use case for network-based file sharing over the Internet through the example of a digital content production that reveals the challenges in inherent this approach to file sharing.

Digital content producers such as photo studios, a digital cinema producers, and video production houses create vast numbers of large files. In the initial step of creating a video, the video producer shoots source videos and stores those files on a storage device. Since many video producers have now started to make high-resolution content such as Full HD (1920 x 1080 pixels), UHD (3840 x 2160 pixels), and 8K (7680 x 4320 pixels), the size of the video files created is often nothing less than massive. For example, a

Table 1.1: Appropriate network-based file sharing techniques with different file sizes, file receiving locations, and the number of file recipients.

File Size	Location	# of recipients	File sharing techniques
Small	Face to Face	single	Adhoc file transmission (AirDrop)
Small	LAN (Short RTT)	single	e-mail, NAS
Small	WAN (Long RTT)	single	e-mail
Small	Face to Face	multiple	NAS, HTTP/FTP Server
Small	LAN (Short RTT)	multiple	NAS, HTTP/FTP Server
Small	WAN (Long RTT)	multiple	Cloud storage, HTTP/FTP Server, CDN
Large	Face to Face	single	*Physical file sharing*
Large	LAN (Short RTT)	single	NAS, HTTP/FTP Server
Large	WAN (Long RTT)	single	Large file delivering service
Large	Face to Face	multiple	NAS, HTTP/FTP Server
Large	LAN (Short RTT)	multiple	NAS, HTTP/FTP Server
Large	WAN (Long RTT)	multiple	HTTP/FTP Server, (**Target of this Thesis**)

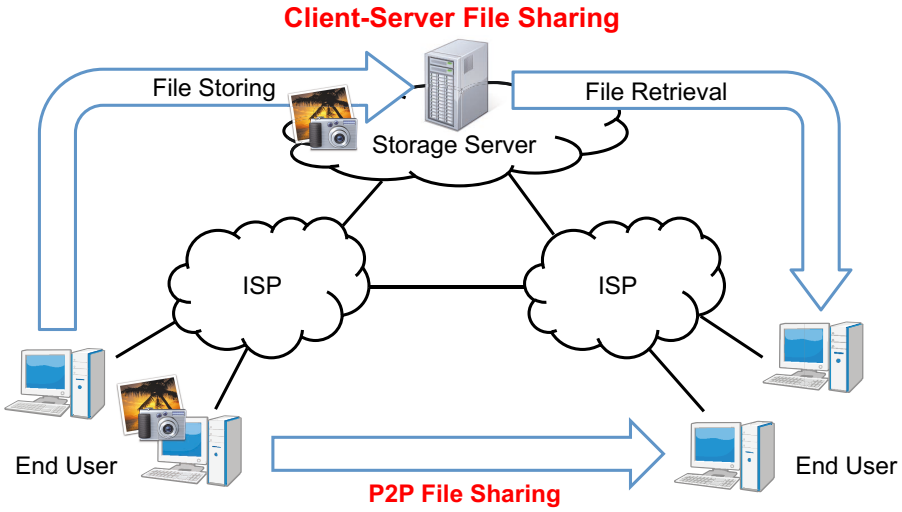


Figure 1.1: Two approaches to network-based file sharing: client-server file sharing and P2P file sharing.

single frame file of uncompressed UHD video is about 24MB when the file is stored in a 24-bit bitmap format. If the frame rate of the video is 30fps (frames per second), the video throughput is about 5.76Gbps, and a 90-minute piece of video content can be as large as 3.8TB. In the second stage of video creation, the video producer edits the video,

adds sound, incorporates visual effects, and so on. These are called post-production processes. In order to create video content rapidly at low cost, video producers usually share their source video files with post-production companies, thus outsourcing most or all of the post-production process. Thus, video producers have to share a large number of very large files with post-production companies as rapidly as possible.

However, it is difficult to achieve low cost, high throughput large file sharing using existing file sharing procedures. First, these files are valuable assets to their owners, so the make copies to ensure wide availability but making extra copies leads to high storage consumption and increased costs. Second, a TCP-based protocol like FTP or HTTP is usually employed for file transmission because TCP is a reliable protocol. However, long-distance transmission using TCP leads to performance degradation. While some TCP implementations (discussed in Section 7) solve this problem, they often demand special operating systems or expensive hardware. Making file caches to shorten transmission distances by using a CDN can avoid the performance degradation found with TCP but creating caches increases the cost of storage.

1.4 Motivation of this study

The previous section showed that achieving high throughput large file sharing over the Internet at low cost remains challenging. The primary motivation of this thesis is to design and implement a low cost high throughput large file sharing system for a global environment. In order to realize this goal, the thesis proposes a storage and retrieval mechanism for large files using globally distributed servers [1]. In the storing procedure of the mechanism, redundant data is added to the original file by FEC, both the redundant data and the original file are divided into chunks, and they are stored to dispersed storage servers. In the delivery procedure, users send a request to all the storage servers that contain the chunks; the servers then send the chunks to users using UDP. Any chunks lost to server failure or packet loss can be recovered by FEC.

1.5 Contributions

One of the two main contributions of this thesis is to demonstrate that UDP with FEC-based chunk storage and delivery in a distributed storage environment can be achieved with both low storage consumption and high throughput, regardless of client locations, by introducing a file storing and retrieval mechanism called DRIP (Distributed chunks Retrieval and Integration Procedure) and Content Espresso, which is a distributed large file sharing system for digital content productions that uses DRIP. Content Espresso is designed to take the actual situation of digital content producers into account, because they often need to store a number of large files and share them rapidly with other people or organizations who may be located at a great distance, even around the world. These requirements demand nothing less than a file sharing system that works on a global scale.

Content Espresso is file sharing system that has a file storage function. Since Content Espresso does not have POSIX-like API and does not provide a writing mechanism to the stored files, it is different from usual file systems.

The other major contribution of this thesis is demonstrating two applications of Content Espresso. The first is designing and implementing Demitasse, which is a network-oriented file-based video playback system that shows the usefulness of Content Espresso. Video playback is an appropriate application for evaluating system performance because it requires large amounts of storage space and high throughput data delivery. This thesis uses uncompressed UHD (3840 x 2160) video content and a minimum throughput of 6Gbps or more at 30fps playback. Demitasse shows that Content Espresso can play uncompressed UHD video at 30fps, so it can be concluded that Content Espresso satisfies the large file sharing demands of digital content producers. The second application involves integrating Content Espresso into SAGE2, a web-based collaboration system, to show that Content Espresso can be used with web-based applications and that it improves the existing file sharing performance of SAGE2. 70% of today's Internet traffic is composed of HTTP-based content. Providing an access interface from HTTP is necessary to deploy Content Espresso in the contemporary world.

1.6 Structure of this thesis

The structure of this thesis is given in Figure 1.2. Chapter 2 introduces the design of Content Espresso, a low cost high throughput large file sharing system for a global environment. Content Espresso is designed based on a storage and retrieval mechanism for large files using globally distributed servers called DRIP. DRIP solves the file transmission performance issues in the global environment by using UDP and FEC. Structurally, Content Espresso is composed of five modules: File Manager, Storage Allocator, Chunk Generator, Cluster Head, and Chunk Server, and Chapter 2 also introduces these modules. Chapter 3 shows the detailed implementation of Content Espresso. The programming model and activity diagram of each module of Content Espresso are shown. Chapter 4 evaluates the file sharing performance of Content Espresso in a global environment by using 79 physical servers, including 72 Chunk Servers. The chapter evaluates Content Espresso from the viewpoint of metadata access performance, file retrieval performance, file storing performance, and system availability. In order to emulate the global environment, the `tc` command is used through the evaluations.

Chapters 5 and 6 demonstrate Content Espresso applications. Chapter 5 introduces Demitasse, a network-oriented UHD video playback system using Content Espresso, which Demitasse uses to store the video component files and a Catalogue System to store the information about relationships among the files. The Demitasse Client, which is a player application, retrieves stored files and plays them back one after another. This chapter shows the design, implementation, and evaluation of Demitasse. Chapter 6 offers an example of the integration of Content Espresso into SAGE2, a web-based collaboration system. This chapter introduces the Relay Server, which provides a web-based

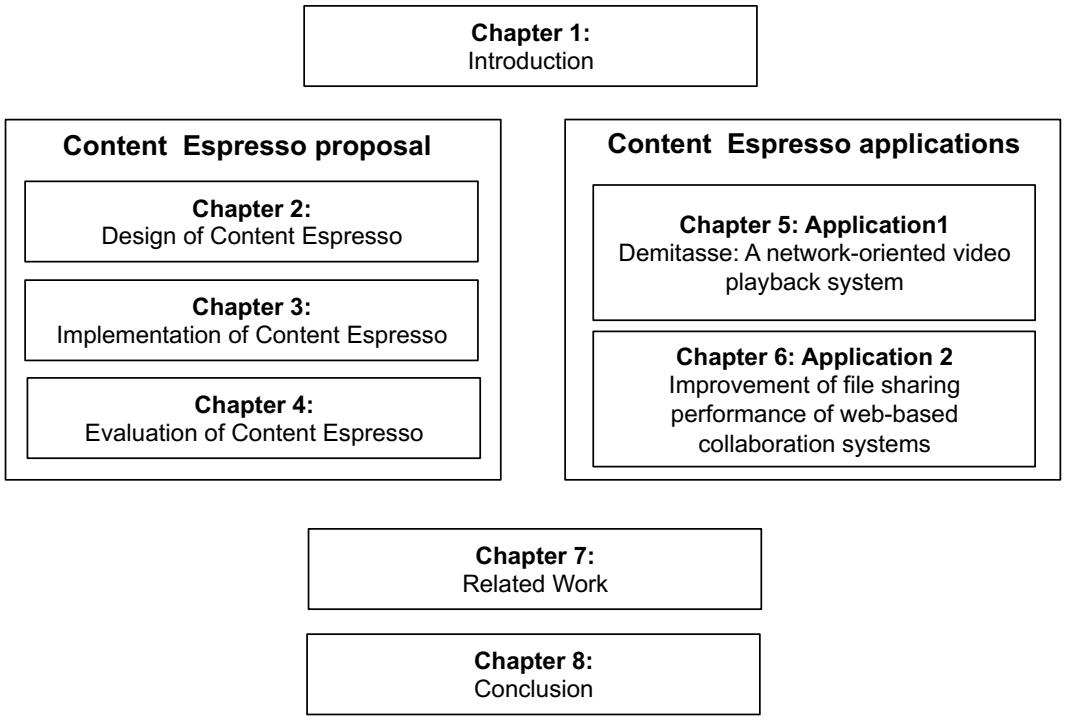


Figure 1.2: Structure of this thesis.

collaboration systems with an HTTP-based access interface for Content Espresso. Finally, Chapter 7 provides a literature review and Chapter 8 concludes the thesis. Chapter 7 clearly shows the differences between Content Espresso and existing systems. It compares them from the viewpoint of the transport layer protocol and as a distributed storage system. Chapter 8 summarizes the thesis, outlines the limitations of Content Espresso, and explores Content Espresso's implications and impact.

The relationship between Content Espresso and Demitasse, which is one of the Content Espresso applications, is as follows. Chapter 2 - 4 introduce Content Espresso. Content Espresso is file sharing system and it provides a high throughput file sharing regardless of the network conditions. Since the requirements of Content Espresso originally come from that of digital content productions, Content Espresso focuses on delivering files with high and stable throughput, and it can deal with any types of file. Chapter 5 introduces Demitasse, a network-oriented UHD video playback system. Demitasse aims at playback video frame files stored in Content Espresso. Although Content Espresso can deliver files with high throughput, it cannot satisfy the requirements of on-demand video distribution and playback. The goal of Demitasse is to retrieve uncompressed UHD frame files from Content Espresso and display them within the duration determined by the frame rate. Therefore, Demitasse is one of the Content Espresso applications for users who store video frame files in Content Espresso.

Chapter 2

Design of Content Espresso

2.1 Background

High throughput and low cost file sharing over the Internet is in great and ever-increasing demand by many industries. It is a particularly urgent need in the digital content sector, which includes video production, film production, and photography of all kinds. When creating digital content, a production company and several post-production companies often work together and must share content files, as shown in Figure 6.2. Each organization has its own solutions for storing content files and managing their metadata.

A typical video production workflow is as follow. First, the production company shoots a high resolution video as sequential frame files and stores them without compression to ensure their reusability. Since one second of video generally consists of 24 to 60 frames, a production company must manage 86,400 to 216,000 frame files for 60 minutes of video content. In addition, the size of each frame file is large. For example, a single frame of uncompressed UHD video (3840 x 2160) is about 24MB in 24-bit bitmap format. If the frame rate of the video is 30fps, 60 minutes of video content can reach up to about 2.5TB. Therefore, the production company has to store a large number of large files. Second, the production company shares these frame files with post-production companies for post-production processes such as video editing, adding a soundtrack, and adding visual effects. Some of the post-production companies are located near the production company, while others are not. They have to retrieve a large number of large files, process them according to the production company's instructions, and return the processed frame files to the production company. The exact files that would be shared vary from company to company. For example, the post-production company A may be assigned to edit chapter 1 of the video and post-production company B to edit chapter 2 of the video. After the processing, processed files are shared as new frame files so as not to overwrite the original frame files. Finally, the production company retrieves all the processed frame files and completes the video production.

Digital content producers need to share large numbers of large files quickly and at low cost. Rapid file sharing can reduce the duration and thus the cost of content production.

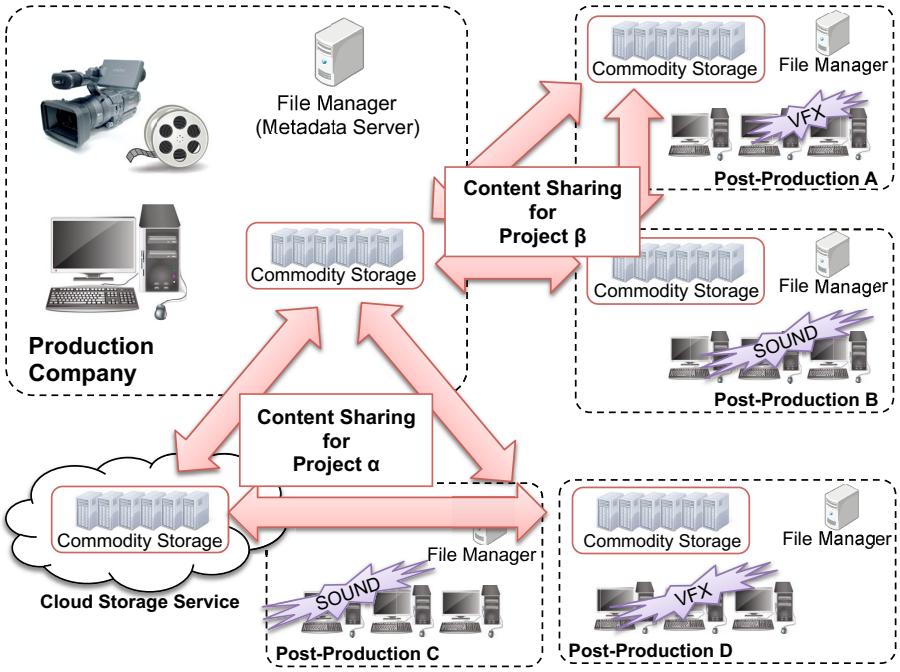


Figure 2.1: Production company and post-production companies work together by sharing content files to make digital content.

Low storage and sharing costs also contribute to reducing the overall cost of a given production. Therefore, digital content producers need high throughput large file sharing in a global environment at low cost. The demand for high throughput large file sharing in a global environment is found not only in the digital content production sector but also in many other industries. Content Espresso is designed to satisfy the demand. Content Espresso has been designed for users with the following characteristics: 1) most of their files are larger than 1MB; 2) they maintain thousands or even hundreds of thousands of files; 3) they share a meaningful number of files with users in other countries; 4) they do not need to retrieve partial files; 5) they prioritize a low cost file sharing system. The remainder of this chapter details requirements, design approach, and actual design of Content Espresso.

2.2 Requirements

The goals that Content Espresso should achieve are defined as follows; high throughput file sharing in a global environment, low storage cost, and secure storage. This section breaks the three goals of Content Espresso down into system requirements.

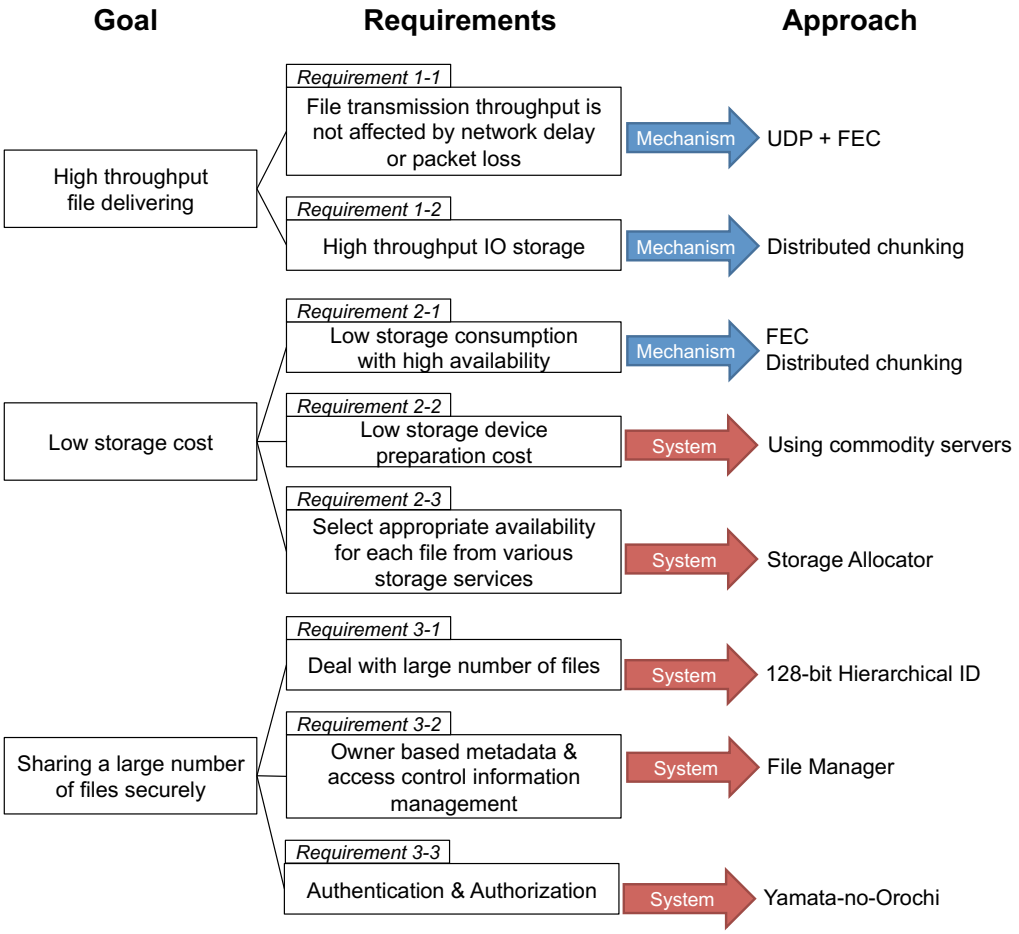


Figure 2.2: Content Espresso goals, requirements, and approaches to meeting them.

2.2.1 High throughput file sharing in a global environment

High throughput file sharing in a global environment is the main motivation behind Content Espresso. A global environment is defined as one in which the locations of file senders and recipients are unknown and could be anywhere in the world. Therefore, network conditions such as the RTT between storage servers and the file retrievers and packet loss among the file senders, storage locations, and file recipients cannot be known in advance. High throughput file sharing in a global environment can be achieved if the following two requirements are satisfied: file transmission retains high throughput regardless of network delays and packet loss (Requirement 1-1) and storage IO with high throughput (Requirement 1-2).

2.2.2 Low storage cost

Low storage cost is the second major motivation behind Content Espresso. Low storage cost can be achieved if these three requirements are satisfied: low storage consumption with high availability (Requirement 2-1), appending appropriate redundancy for each stored file (Requirement 2-2), and low storage device preparation and management cost (Requirement 2-3). In order to achieve high availability, the system usually appends redundant data to the stored data, which causes high storage consumption. Although redundancy is necessary to ensure high availability, the ideal system would adopt a technique that maximally reduces redundancy while keeping availability as high as possible. In addition, the system should be able to manage files while taking specific required availability levels into consideration. The specific availability level required depends on the files. If a system does not allow changes in availability level for each file with an approach such as redundant array of independent disks (RAID) [6], all files must be stored at the highest availability level, which wastes storage resources and increases storage costs. To achieve minimal storage consumption with the availability required, the system should be able to select an appropriate availability level for each file. Storage preparation and management costs also must be considered, and the system should be designed to keep those costs low.

2.2.3 Sharing a large number of files securely

Sharing a large number of files globally requires that files have global unique identifiers (Requirement 3-1). Content Espresso assumes that a large number of users share a large number of files. A global-scale file sharing system requires a global unique file identifier. In the digital cinema industry, about 700 films are released every year in year in the US and Canada [7]; a single film usually consists of more than 100,000 files. Thus, a system must have a global unique file identifier and user identifier that have enough characters to identify all these content files and users.

The security of the stored files is also an important aspect of Content Espresso, because these files are their owners' assets. Although there are many aspects to storage security, Content Espresso focuses on preventing file leakage though access by unauthorized users. Thus, maintaining the metadata and access control information securely (Requirement 3-2) and authenticating and authorizing users (Requirement 3-3) are essential. The system should be able to manage metadata and access control information entirely within the file owner's organization. Because of the need to share files and the sheer shale of operations, the system should also support multi-domain authentication and authorization to guarantee that only legitimate users can access the files.

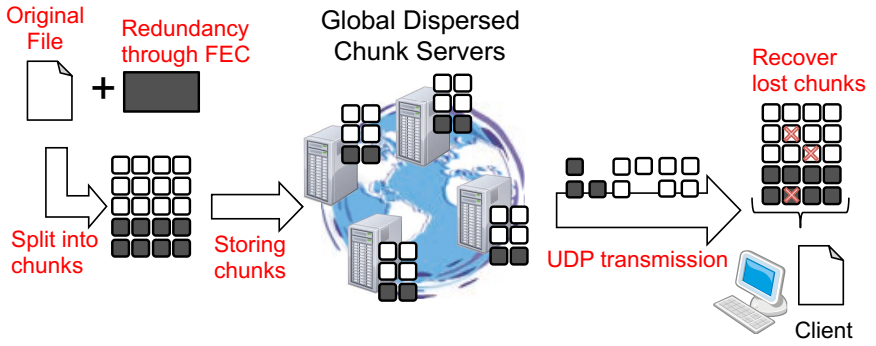


Figure 2.3: Overview of DRIP, which achieves low cost data storage and high throughput transmission[1].

2.3 Approach of Content Espresso

Content Espresso is designed to satisfy all the requirements laid out in the previous section. The design approach of Content Espresso can be divided into two elements; the design approach for storage and retrieval purposes, and the design approach of the overall system. Figure 2.2 depicts the requirements and corresponding elements of the approach.

2.3.1 Approach as a storage and retrieval mechanism

In order to realize Requirement 1-1, Requirement 1-2, and Requirement 2-1, the storage and retrieval mechanism DRIP (Distributed chunks Retrieval and Integration Procedure) is used [1]. The key techniques of DRIP are to append redundant data to the original file using FEC, splitting the original file and redundant data into chunks, sending them to dispersed storage servers, and delivering the stored chunks with UDP. Figure 2.3 offers an overview of DRIP.

In the first step, a file is split into blocks and redundant data added by FEC to each block. Then, both the file data and the redundant data are divided into chunks that are sent to storage servers called *Chunk Servers* all over the world, with a sequence number assigned to each chunk. The size of an FEC block and the amount of redundant data are determined by the users who are storing the file. DRIP can work with any FEC algorithm, but the implementation shown here utilizes a low-density generator matrix (LDGM) coding [8] for FEC, because LDGM has an advantage over Reed-Solomon coding in lost bit recovery time with large-sized blocks [9]. The sequence number assigned to each chunk describes the position of the chunk in the file. When a *Client* retrieves a stored file, the Client sends a data retrieval request to the Chunk Servers. The Chunk Servers send the chunks that compose the requested file to the Client through UDP. The Client receives the chunks one after another and reorders them by each chunk's sequence

number. Even if some chunks are not delivered because of packet loss in the network or temporary server failure, the Client can recover lost chunks using the redundant data.

2.3.2 Approach of the overall system

In order to satisfy Requirement 2-2, Requirement 2-3, Requirement 3-1, and Requirement 3-2, the approach to designing Content Espresso adopted principles detailed in the paragraphs below.

First, the system uses the storage and retrieval mechanism proposed in the previous subsection and commodity storage servers called *Chunk Servers*. The storage and retrieval mechanism contributes to low storage cost and high throughput file transmission using multiple Chunk Servers. While Content Espresso utilizes multiple globally dispersed Chunk Servers, establishing globally dispersed storage servers usually carries a high cost. The system should provide high IO throughput by using inexpensive commodity servers that can sometimes be low-IO throughput servers. Content Espresso uses commodity storage servers that organizations own or rent from existing data center services as Chunk Servers to reduce storage preparation costs. The system aggregates Chunk Servers into a Chunk Server Cluster in each organization, which also lowers costs.

Second, the system introduces a *Storage Allocator* that selects and aggregates the various Chunk Server Clusters and configures storage services to meet user demands in terms of disk IO, storage space, availability, and price. For example, a Storage Allocator selects a sole, highly available Chunk Server Cluster and configures a highly available storage service at a single location to store files that require high availability. Content Espresso has multiple Storage Allocators, each of which can build multiple storage services to meet differing user demands without the need to buy or rent additional hardware. Figure 2.4 shows the positions of the Chunk Servers, the Chunk Server Clusters, the Storage Allocator, and the storage services.

Third, the system introduces a *File Manager* that is established for each organization to manage file metadata and user information inside that organization. Metadata includes the file owner's identifier, access control information, selected storage service, etc. The File Manager generates and manages *Local User IDs* and *Local File IDs* to manage users and file metadata. The Local User ID is a user identifier assigned to each user, while the Local File ID is a unique file identifier assigned to the file by the user who is storing it on Content Espresso. The File Manager that assigns the Local User ID to a user is called the *Home File Manager* of the user. The length of a Local User ID is 32 bits; that of a Local File ID is 96 bits. The length of these IDs is long enough to manage users and content files inside even very large organizations. Figure 2.5 shows the relationship among File Managers, Storage Allocators, and storage services.

Fourth, the system utilizes a *Global File ID (GFID)*, a globally unique hierarchical 128-bit file identifier that enables sharing a very large number of files among several or even many organizations. Content Espresso assigns a global unique 32-bit *File Manager ID* to each File Manager and ensures the global uniqueness of the GFID and the *Global*

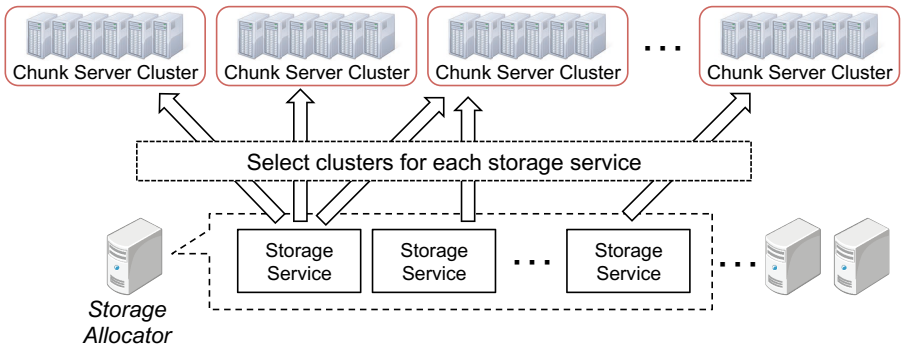


Figure 2.4: Chunk Servers configured as a Chunk Server Cluster in each organization; a Storage Allocator configures multiple storage services by selecting Chunk Server Clusters from a variety of options.

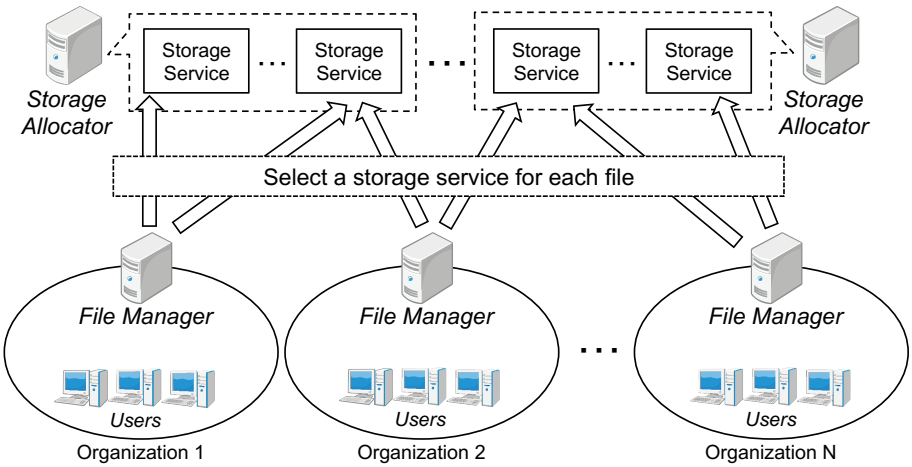


Figure 2.5: Each organization has a File Manager that manages file metadata, including selecting the storage service.

User ID (GUID) by introducing a hierarchical identifier structure, as shown in Figure 2.6. The GFID is composed of the 32-bit File Manager ID and the 96-bit Local File ID. The GUID is composed of the 32-bit File Manager ID and the 32-bit Local User ID. Since the system ensures that the File Manager ID is globally unique, the GFID and the GUID are guaranteed to be globally unique as well.

Finally, the system uses the Yamata-no-Orochi [10, 11] authorization and authentication system to achieve secure file sharing in a multi-domain environments.

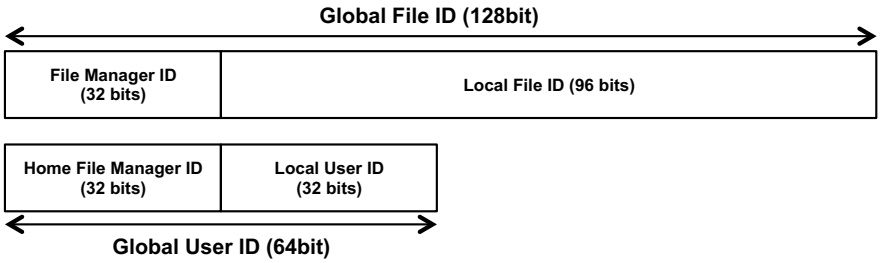


Figure 2.6: A Global File ID consists of a File Manager ID and a Local File ID; a Global User ID consists of a Home File Manager ID and a Local User ID.

2.4 System design

2.4.1 Content Espresso modules

Content Espresso consists of four main modules: Chunk Server, Storage Allocator, File Manager, and Client. In addition, there are two sub-modules: Cluster Head and Chunk Generator. Figure 2.7 shows the whole architecture of Content Espresso.

Chunk Server and Cluster Head

The Chunk Server stores chunks to the local file system and sends them to the Client using UDP upon receiving a file retrieval request from users. Content Espresso has two types of chunks: data chunk and parity chunk. The data chunk is a chunk that composes the original file. The parity chunk is a chunk that composes the redundancy data produced by the FEC coding. The data and parity chunks are gathered and stored as a data chunk file and a parity chunk file at the Chunk Servers, respectively.

When the chunks are stored, 8-byte header including a Sequence Number is appended to each chunk. Figure 2.8 shows the chunk management architecture. The chunk size should be less than the MTU (Maximum Transmission Unit) of an IP packet because Content Espresso recovers lost chunks caused by packet loss. Content Espresso utilizes 1,272-byte chunk size taking the block size of the local file system into consideration. Each chunk is stored as 1,280-byte data (header + chunk) and 16 chunks are stored as a 20,480-byte file, which is 5 times of typical block size 4,096 bytes.

The Cluster Head is a sub module of the Chunk Server and exists at least one for each Chunk Server Cluster as the access interface for the Chunk Servers. The Cluster Head relays a file retrieval request to all the Chunk Servers inside the cluster using multicast to reduce the request time lag and the request traffic. Since the Cluster Head does not have any information about which Chunk Server stores which chunks, it relays all the retrieval requests to all the Chunk Servers that belong to the same cluster. Upon receiving a file storage request, the Cluster Head receives chunks from the Chunk Generator, which is a sub module of the Storage Allocator, and sends them to the Chunk Servers with the

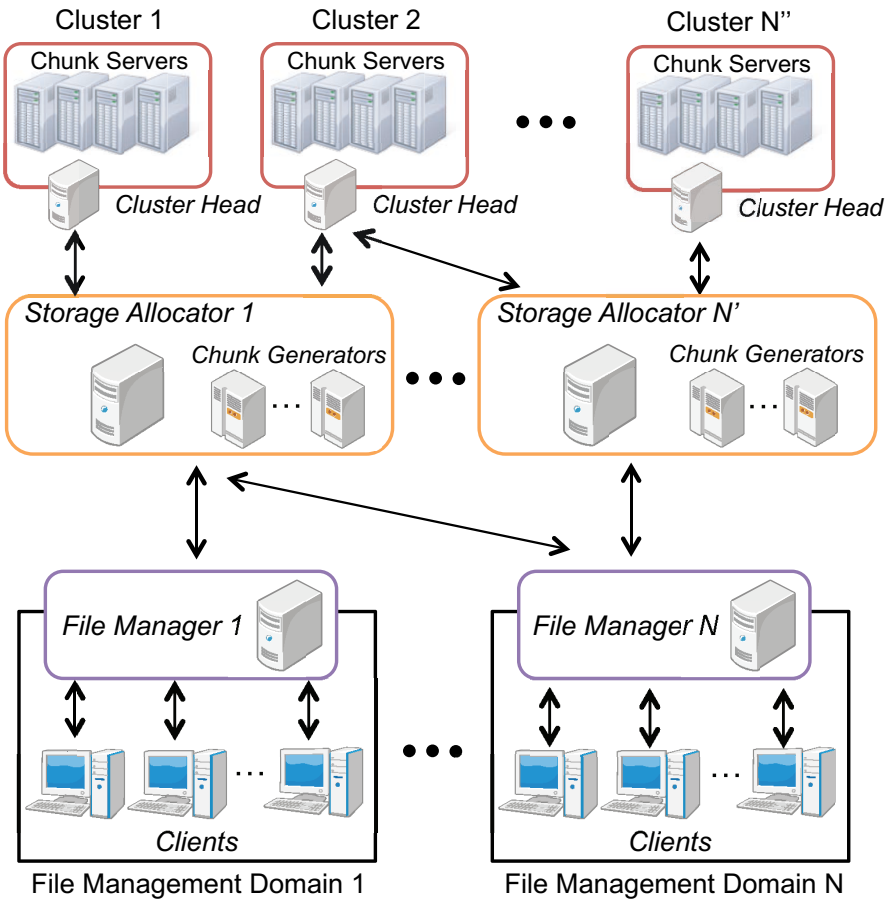


Figure 2.7: Content Espresso consists of a Chunk Server, a Cluster Head, a Storage Allocator, a Chunk Generator, a File Manager, and a Client.

round-robin algorithm to avoid burst loss when some Chunk Servers fail.

Storage Allocator and Chunk Generator

Content Espresso has multiple Storage Allocators that aggregate multiple Chunk Server Clusters and the Storage Allocator provides various storage services to users. The Storage Allocator has a database containing two tables: *chunk_table* and *cluster_table*. It manages the clusters and placement of chunks. The *chunk_table* stores the mapping of the Global File ID and the cluster in which the chunks of the file are stored. The *cluster_table* stores the information of the clusters such as the IP address of the Cluster Head. For each retrieval request from the File Manager, the Storage Allocator accesses the database, gets the IP address list of the Cluster Head whose cluster stores the requested file, and relays the request to the Cluster Heads.

The Storage Allocator has multiple *Chunk Generators*, sub modules of the Storage

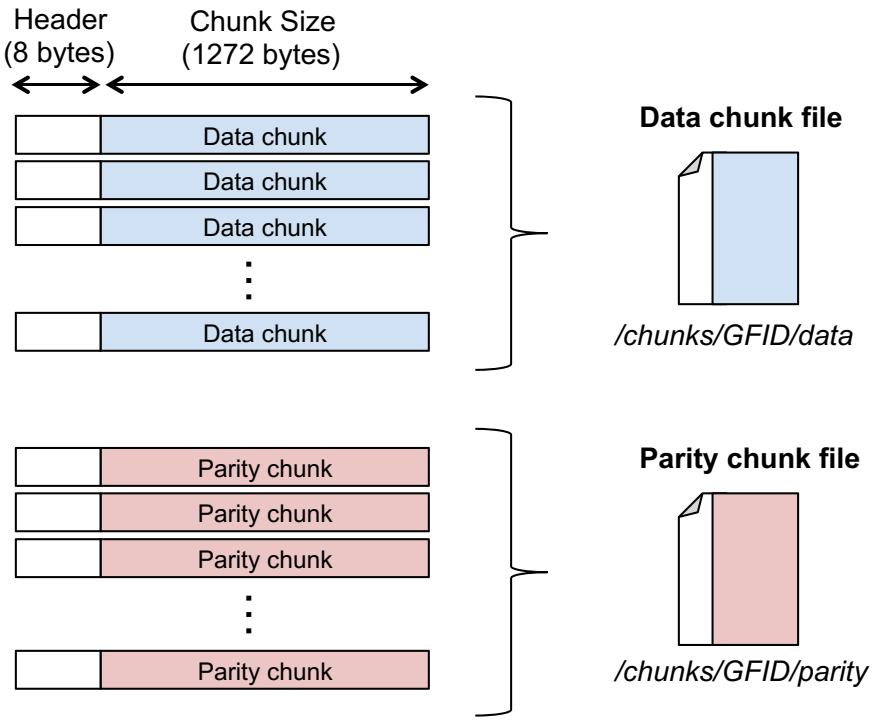


Figure 2.8: Each Chunk Server stores chunks and their headers as a data chunk file and a parity chunk file to its local file system.

Allocator. The Chunk Generator receives files from the Client, appends redundancy data to the files, splits them into chunks, and sends them to the Cluster Head when a file storage request comes from the Storage Allocator. The Chunk Generators are located over the world to reduce the file storage time because they use TCP for file storage. The Storage Allocator assigns the Chunk Generator to each file storage request in terms of the workload and location of the Chunk Generator.

The Storage Allocator manages and monitors the Chunk Server Clusters. When the Storage Allocator receives a Chunk Server failure report from the Cluster Head, it assigns a Chunk Generator and the Chunk Generator retrieves available chunks from other Chunk Servers in the cluster, recovers the chunks stored in the failed Chunk Server using FEC, and restores them to the Chunk Servers.

File Manager

The File Manager manages the file metadata and the user information for access control using a database. The database has three tables: *file_table*, *fec_table*, and *sa_table* as shown in Table 2.1 to 2.3. The file_table stores the file metadata. The fec_table stores the information of available FEC algorithms. The sa_table stores the information of the Storage Allocators such as their IP addresses.

Table 2.1: File Manager Database: *file_table*.

Field	Description
file_id	Global File ID
sa_id	Storage Allocator ID
filesize	Original file size
fec_filesize	Total file size after generating parity
fec_id	FEC ID
owner_id	Global User ID of file owner
chunk_size	Chunk size
data_block_height	H_{Data} (Explained in Subsection 2.4.2)
parity_block_height	H_{Parity} (Explained in Subsection 2.4.2)

Table 2.2: File Manager Database: *sa_table*.

Field	Description
sa_id	Storage Allocator ID
hostname	Host name of Storage Allocator

Table 2.3: File Manager Database: *fec_table*.

Field	Description
fec_id	FEC Identifier
fec_info	FEC Information

For each retrieval request from the Client, the File Manager assigns the *Session ID*, accesses the database, gets the IP address of the Storage Allocator, and relays the request to the Storage Allocator. The File Manager also authenticates Clients and authorizes them for each access. Content Espresso utilizes the Yamata-no-Orochi authentication and authorization system.

Client

The Client issues a file store or retrieval request to Content Espresso through the File Manager. In the file store process, the Client sends a file to one of the Chunk Generators. In the file retrieval process, the Client receives chunks from the Chunk Servers and recovers lost chunks using the FEC code. The Client has user authentication information used for authentication by the File Managers.

2.4.2 FEC algorithm and FEC block size

Content Espresso allows users to select a FEC algorithm and FEC block size. It supports LDGM (Low Density Generator Matrix) as the FEC algorithm in the current implementation. LDGM utilizes the same generator matrix file for generating redundancy data and

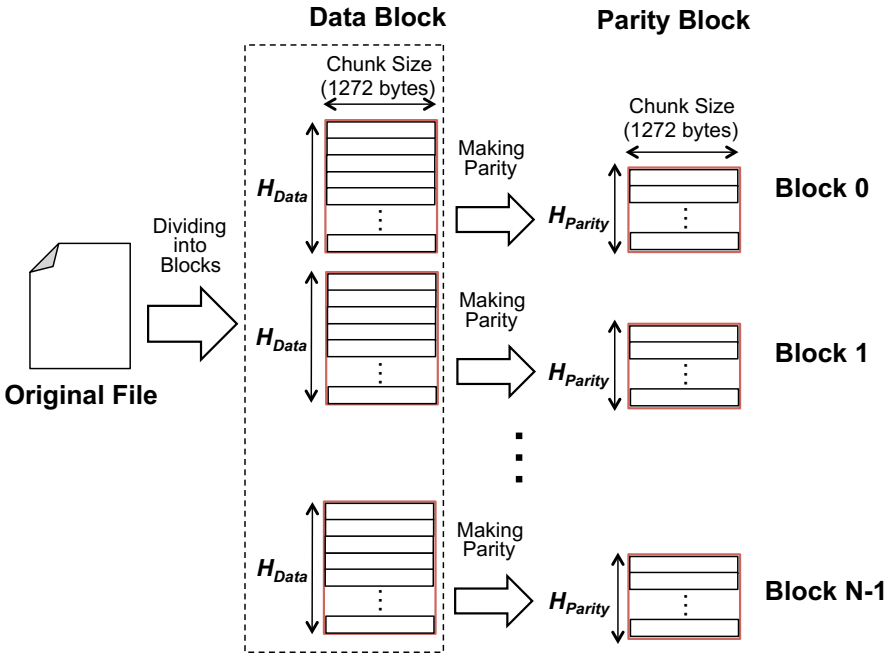


Figure 2.9: An original file is divided into data blocks and parity blocks generated by FEC. H_{Data} and H_{Parity} can be configured by Clients.

recovering lost chunks. The matrix file is dependent on FEC block size and redundancy rate.

Figure 2.9 depicts the relationship among the original file, the data block, and the parity block. A FEC block composed of the data chunks is called a *data block* and that composed of parity chunks is called a *parity block*. Users can select the size of data block and the redundancy rate when they store files. The data block size is calculated based on the chunk size and the number of chunks of one data block. The number of chunks of a data or parity block is defined as the data block height (H_{Data}) and the parity block height (H_{Parity}), respectively. Thus, the data block size can be calculated as $ChunkSize \times H_{Data}$.

The data block height in LDGM should be more than 1,000 because less than 1,000 data block height causes performance degradation in lost chunk recovery. The redundancy rate can be calculated as H_{Parity}/H_{Data}. Thus, in case that the data block height is 1,000 and the parity block height is 200, the redundancy rate is $200/1,000 = 0.2$. Since the FEC parameter is defined by the chunk size, the data block size, and the redundancy rate, the File Manager stores the FEC parameter as the metadata for each file.

Figures 2.10 and 2.11 depict the procedure of making parities and recovering lost chunks when Content Espresso stores and retrieves the file. After the parity data is appended at the Chunk Generator, the Chunk Generator stores the chunks to Chunk Servers by adding the sequence number by using the round-robin algorithm.

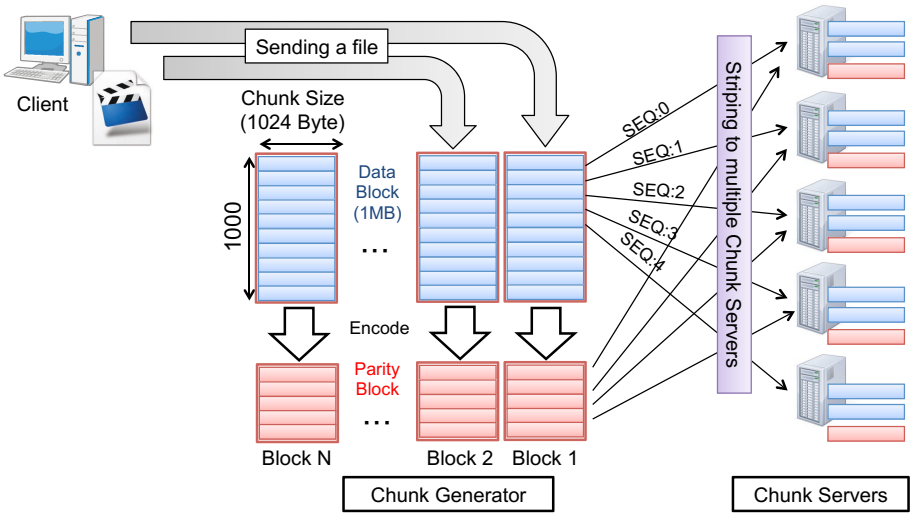


Figure 2.10: The procedure of making parities and distributing to multiple Chunk Servers.

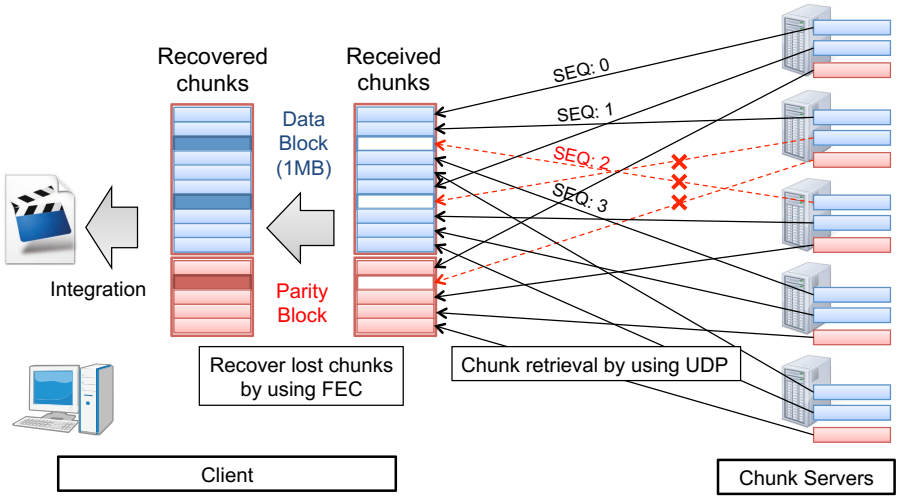


Figure 2.11: The procedure of retrieval chunks from Chunk Servers and recovering lost chunks.

2.4.3 System availability

Content Espresso ensures the system availability by using FEC and distributing chunks to the multiple Chunk Server Clusters and Chunk Servers. Figure 2.12 depicts the Content Espresso and its possible causes of chunk lost.

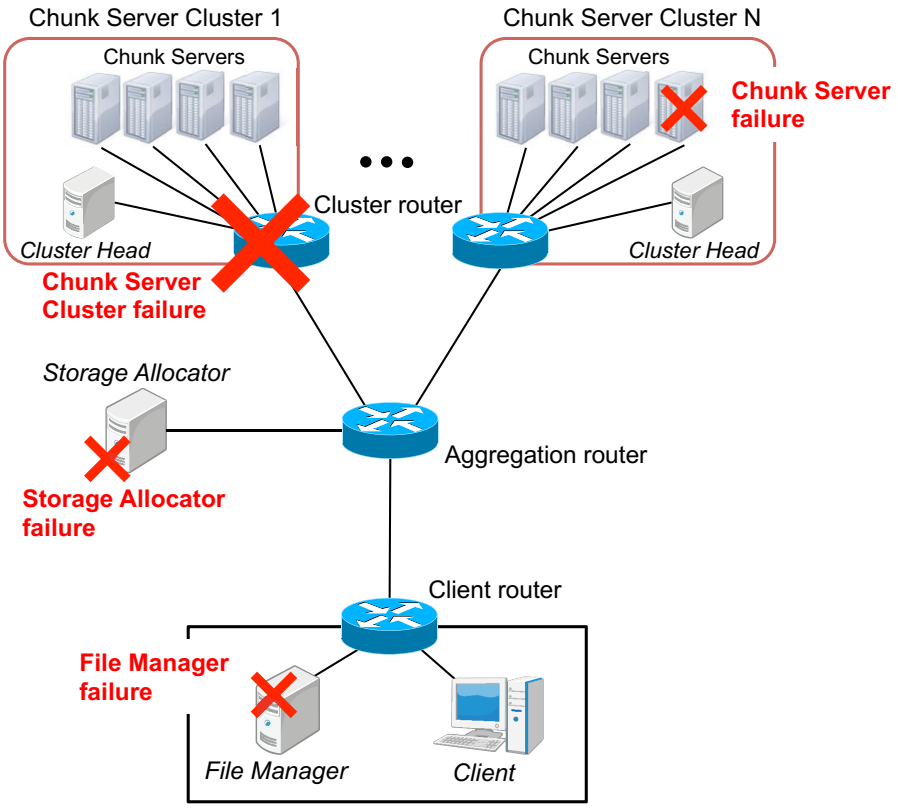


Figure 2.12: Content Espresso's possible causes of chunk lost can be classified into four: Chunk Server failure, Chunk Server Cluster failure, Storage Allocator failure, and File Manager failure.

Chunk Server failure

Content Espresso assumes that each Chunk Server is composed by commodity hardware. Thus, software failure and hardware failure are main causes of the Chunk Server failure. Software failure of Chunk Server consists of unexpectedly OS shutdown, Chunk Server process is killed, software bug, and so on. Hardware failure of Chunk Server consists of power module failure, HDD failure, NIC (Network Interface Card) failure, and so on. In order to tolerate the Chunk Server failure, Content Espresso distributes chunks to multiple Chunk Servers by using round-robin algorithm.

Chunk Server Cluster failure

A Chunk Server Cluster is located in a single location. Thus, the power outage or network failure at the location causes the Chunk Server Cluster failure. In addition, each Chunk Server Cluster has a Cluster Head, which is the interface of the Cluster. Thus, the Cluster Head failure also causes the Chunk Server Cluster failure. In order to tol-

erate the Chunk Server Cluster failure because of the power outage or network failure at the location, Content Espresso distributes chunks to multiple Chunk Server Clusters by using round-robin algorithm. The number of necessary clusters is dependent on the redundancy rate of store files. In case that Content Espresso stores the files with 20% redundancy rate, more than ten clusters are necessary, in general.

File Manager and Storage Allocator failure

A File Manager manages the metadata of stored files and users have to access the File Manager before starting file retrieval. A Storage Allocator has the mapping of stored files and the used Chunk Server Clusters. Thus, all users cannot access the files that are managed in the File Manager or the Storage Allocator when the File Manager or the Storage Allocator failures, respectively. Content Espresso does not have the mechanism to tolerant the File Manager failure and the Storage Allocator failure. In practically, existing solutions such as database replication and preparing hot standby server can be used to ensure the File Manager and Storage Allocator availability.

2.5 File access procedure

This subsection shows the procedures of the file retrieval and storing in Content Espresso. Messages among each module are sent by TCP except between the Cluster Head and the Chunk Server and between the Chunk Server and the Client in the file retrieval. TCP connections are established when the process of each module starts to reduce the overhead of connection establishment.

2.5.1 File retrieval sequence

The file retrieval procedure consists of three phases: *Authentication Phase*, *Metadata Retrieval Phase*, and *File Retrieval Phase*. Figure 2.13 shows the details of these phases.

Authentication Phase

A Client accesses the File Manager that manages the metadata of the target file and sends an AUTHSYSTEM_REQUEST message. The message contains the identifier of the authentication system that the Client supports and the File Manager responds to the request using an AUTHSYSTEM_RESPONSE message. Then, the Client sends an AUTHENTICATION_REQUEST message to the File Manager and the File Manager authenticates the Client. The authentication procedure depends on authentication types. In the current implementation, Content Espresso supports Yamata-no-Orochi as the user authentication and authorization system.

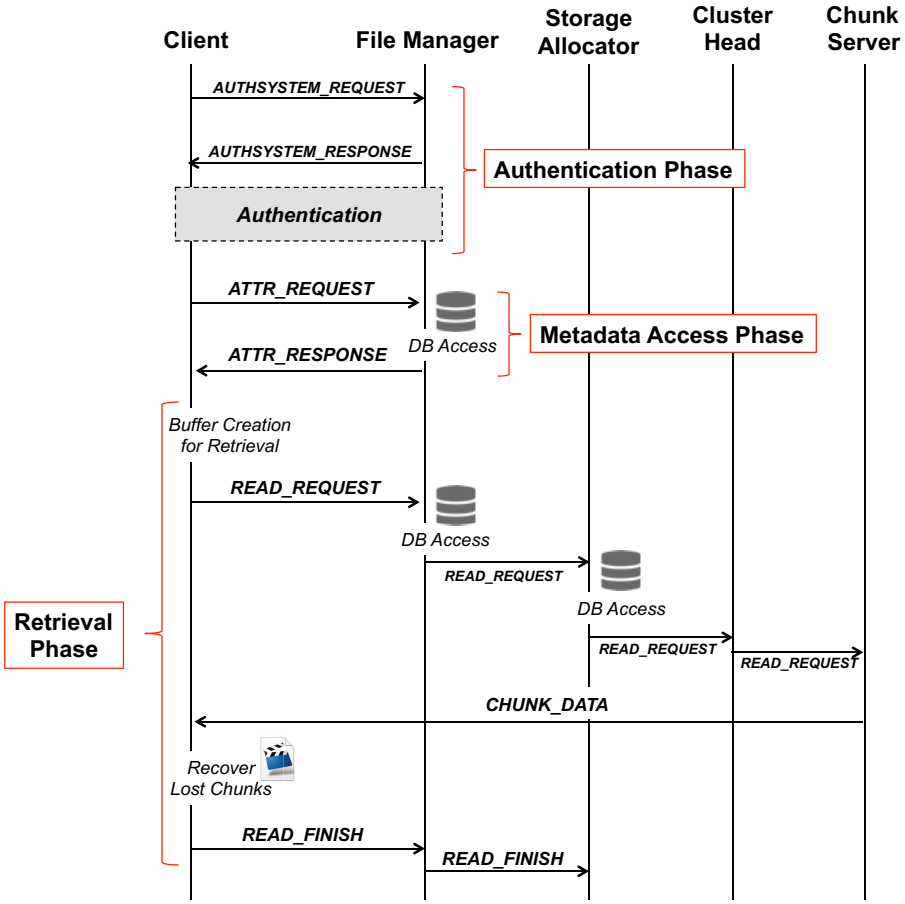


Figure 2.13: The file retrieval sequence consists of the Authentication Phase, the Metadata Access Phase, and the Retrieval Phase.

0								1								2								3							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type								Code								Length															
Session ID (128 bits)																															
Global File ID (128 bits)																															

Figure 2.14: Message format of ATTR_REQUEST.

Metadata Retrieval Phase

The Client sends an ATTR_REQUEST message to the File Manager that manages the metadata of the target file. Figure 2.14 depicts the packet format of the ATTR_REQUEST message. When the File Manager receives the request message from the Client, the File Manager authorizes the request, accesses the database to obtain the metadata of the requested file, and sends the results to the Client

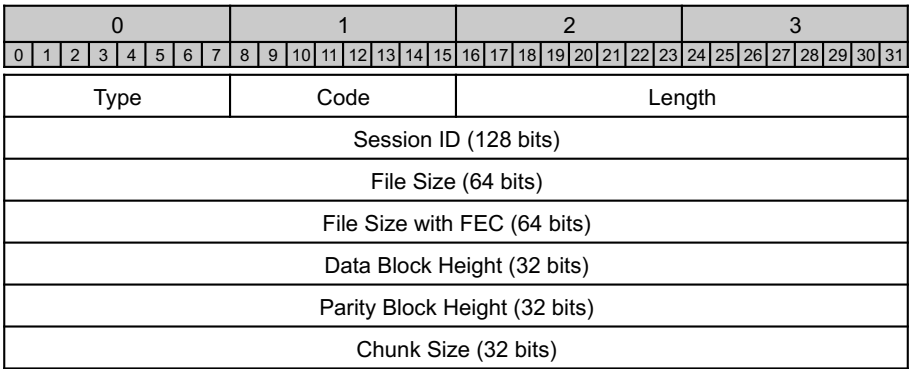


Figure 2.15: Message format of ATTR_RESPONSE.

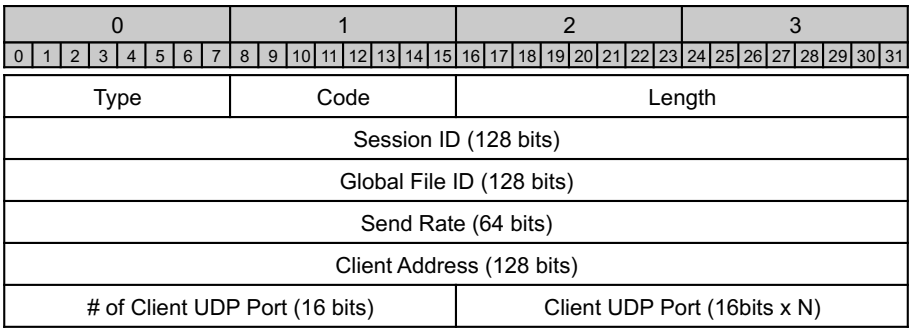


Figure 2.16: Message format of READ_REQUEST.

in an ATTR_RESPONSE message. Figure 2.15 depicts the packet format of the ATTR_RESPONSE message.

File Retrieval Phase

The Client sends a READ_REQUEST message to the File Manager. Figure 2.16 depicts the packet format of the READ_REQUEST message. When the File Manager receives the message from the Client, it authorizes the request, accesses the database to obtain the IP address of the Storage Allocator that manages the requested file, and relays the request message to the Storage Allocator. The Storage Allocator accesses the databases to obtain the number of Chunk Servers and the cluster that stores the chunks of the file, calculates the chunk transmission rate, and sends the READ_REQUEST message to the Cluster Heads.

After the Cluster Heads receives the request, they relay it to the Chunk Servers using multicast. Upon receiving the request, each Chunk Server transmits the data chunks and the parity chunks corresponding to the Global File ID to the Client IP address using UDP. The Client receives the chunks and recovers the undelivered chunks after a timeout. If non-recovered blocks exist, the Client sends the READ_REQUEST message to the File Managers again. Otherwise, the Client sends a READ_FINISH message to the File

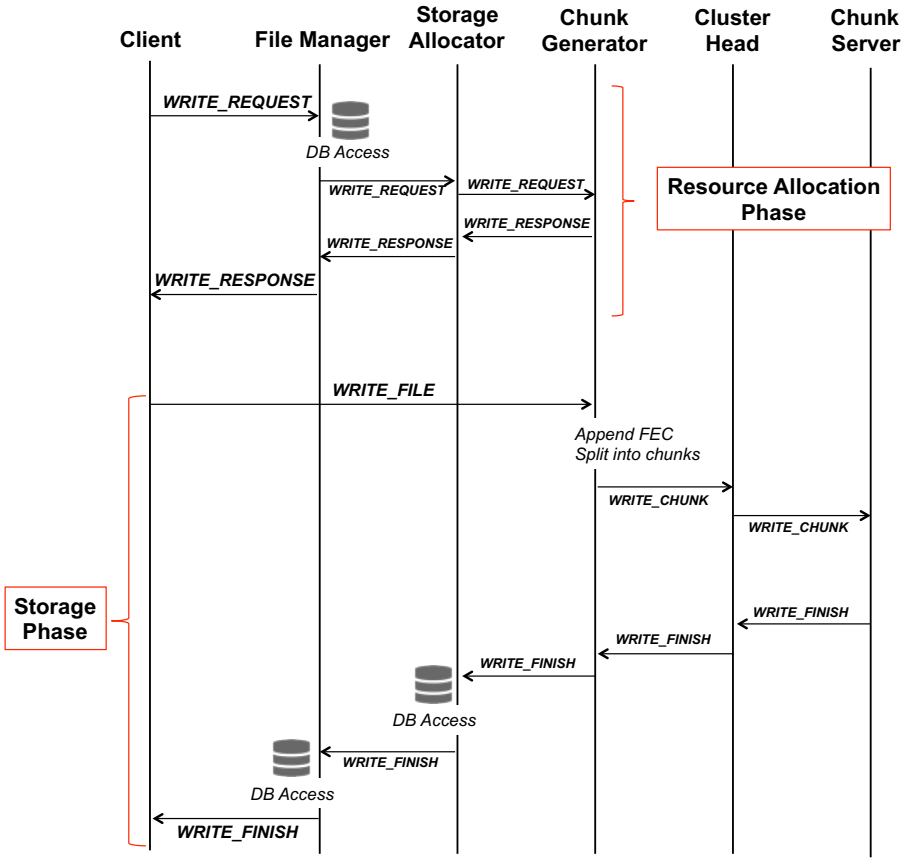


Figure 2.17: The file storing sequence consists of the Authentication Phase, the Resource Allocation Phase, and the Storage Phase. The Authentication Phase is the same as that of the file retrieval sequence.

0								1								2								3							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Type								Code								Length															
Session ID (128 bits)																															
Global File ID (128 bits)																															
File Size (64 bits)																															
Client Address (128 bits)																															
Storage Service & FEC Parameters																															

Figure 2.18: Message format of WRITE_REQUEST.

Manager. The format of the READ_FINISH message is the same as the first 20 bytes of the READ_REQUEST message.

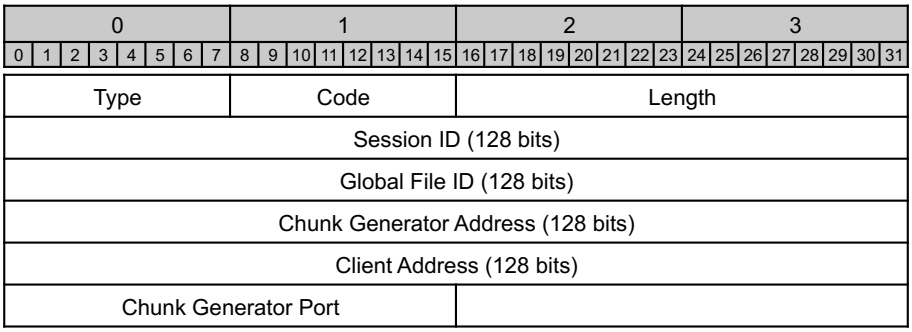


Figure 2.19: Message format of WRITE_RESPONSE.

2.5.2 File storage sequence

The file storage procedure consists of three phases: the *Authentication Phase*, *Resource Allocation Phase*, and *Storage Phase*. Figure 2.17 shows the details of these phases. The Authentication Phase is the same as that in the file retrieval sequence.

Resource Allocation Phase

The Client sends a WRITE_REQUEST message to the Home File Manager. Figure 2.18 depicts the packet format of the WRITE_REQUEST message. The WRITE_REQUEST message contains the File ID, the File Size, the Client Address, the type of storage service, and the FEC parameters. When the File Manager receives the message, it accesses the database to reserve the Global File ID and sends the request message to the Storage Allocator. If the Global File ID in the message is zero, the Global File ID is automatically assigned by the File Manager.

The Storage Allocator assigns the request a storage service and a Chunk Generator to store content files and returns a WRITE_RESPONSE message to the Client via the File Manager. The WRITE_RESPONSE message contains the File ID, the Chunk Generator Address, and the Chunk Generator port number. Figure 2.19 depicts the packet format of the WRITE_RESPONSE message.

Storage Phase

After the Client receives the WRITE_RESPONSE message from the File Manager, it sends the file data to the Chunk Generator assigned by the Storage Allocator. The Chunk Generator splits the file data into FEC data blocks and generates parity blocks using the FEC parameter indicated by the Client. Then the Chunk Generator splits each block into chunks and sends them to each Cluster Head with assigning the sequence number to each chunk. The Cluster Head also distributes the received chunks to Chunk Servers inside the same cluster.

After finishing storing chunks, the Cluster Head returns a WRITE_FINISH message to the Storage Allocator via the Chunk Generator. The Storage Allocator stores the

Table 2.4: Client API functions.

Function	Argument
int loginToFileManager	(**int** authType)
int sendAttrRequest	(**FILE_ID** fileId)
int allocateFecBuffer	(**EspressoBuffer&** eb)
int allocateRecvBuffer	(**char*** recvBuffer)
int recvContent	(**EspressoBuffer*** eb, **LdgmDecode*** ldgm, **unsigned char*** recvBuffer, **FILE_ID** fileId, **uint64_t** sendRate, **uint16_t** nUdpPort)
int setWriteInfo	(**uint32_t** nCluster, **uint32_t** nCs, **uint32_t** chunkSize, **uint32_t** dBlockHeight, **uint32_t** pBlockHeight)
int sendContent	(**FILE_ID&** fileId)

Global File ID and utilized clusters' information to the database and sends the finish message to the File Manager. When the File Manager receives the WIRTE_FINISH message, it stores the total file size including FEC data and relays the message to the Client. The format of the WRITE_FINISH message is the same as the first 20 bytes of the WRITE_REQUEST message.

2.6 Client API

Content Espresso provides an API written in C++ to access the File Manager to store and retrieve files. Table 2.4 describes the main functions of the API. `loginToFileManager()` is the function to login to the Home File Manager. `sendAttrRequest()` is the function to send a request to the File Manager to retrieve the metadata of the requested file. `allocateRecvBuffer()` is the function to allocate memory for receiving arrival chunks. `allocateFecBuffer()` is the function to allocate memory to reorder arrived chunks and recover lost chunks. `recvContent()` is the function to send a request to the File Manager to retrieve the chunks. `sendContent()` is the function to send a request to the File Manager to store the files.

Chapter 3

Implementation of Content Espresso

All Content Espresso modules are implemented in C++ in CentOS 6.0. The File Manager and Storage Allocator use MariaDB 10.1.8 [12] as the database management system. In order to improve performance, Content Espresso modules have multiple threads, and to prevent those multiple threads from accessing critical sections simultaneously, Content Espresso modules use the `std::mutex` and `std::condition_variables` that are provided in C++11.

3.1 Implementation model of each module

Content Espresso assumes that many Clients request file retrieval and storage simultaneously. Therefore, the File Manager, Storage Allocator, Chunk Generator, Cluster Head, and Chunk Server all need to be designed and implemented to deal with large numbers of concurrent requests. In order to achieve parallelization in servers, many implementation models have been proposed. Content Espresso adopts a thread pool and `epoll()` approach because the thread pool and `epoll()` model has advantages of response time and resource utilization ratio. The number of threads in the thread pool must be chosen by the operator before starting the process that takes the available resources into account.

Figure 3.1 describes the thread pool and `epoll()` implementation model, which consists of multiple worker threads and an accept thread. The accept thread accepts the TCP connection request from another module. When the accept thread accepts the request, it registers the accepted socket descriptor in the epoll buffer. Each worker thread waits until the message arrives by using `epoll_wait()`. When the request comes to the registered socket, `epoll_wait()` returns and announces the arrival of message to the process. `epoll()` has two types of triggers: a level trigger and an edge trigger. These triggers are conditions for `epoll_wait()` to announce the arrival of messages to the process; users have to select one of the two types of trigger. The level trigger is used when the receiving buffer is not empty, while the edge trigger triggered when a new packet arrives. Content Espresso modules adopt the edge trigger so that receiving arrival messages are not forgotten.

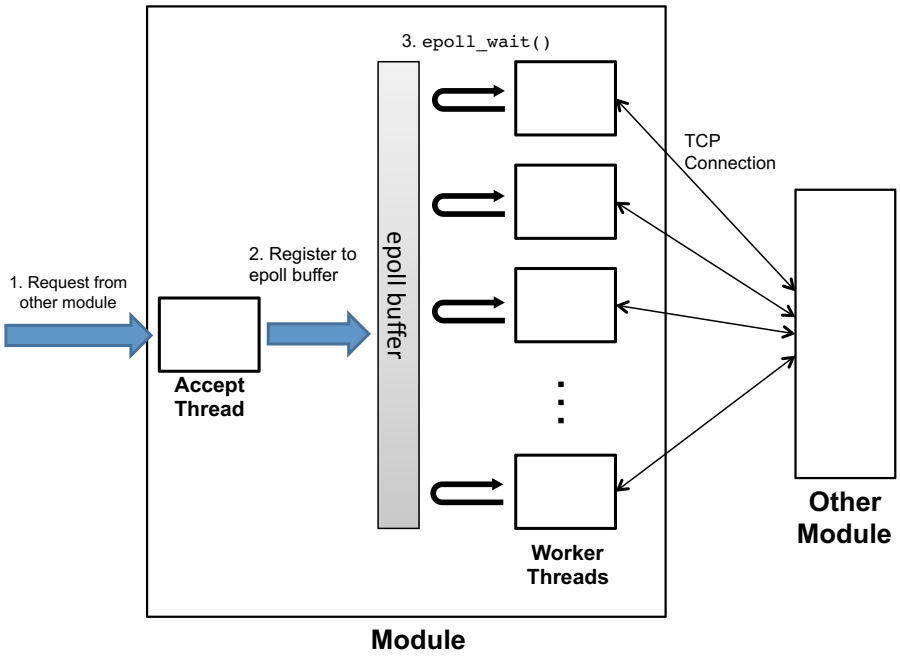


Figure 3.1: Model of epoll and thread pool.

Table 3.1: File Manager's required parameters.

Type	Name	Description
uint16_t	fm_port	Port number of File Manager
uint32_t	sa_id	Identifier of Storage Allocator to store files
uint16_t	sa_port	Port number of Storage Allocator
uint32_t	thread_num	Number of worker threads
std::string	log_file	Path to the log file

3.2 File Manger

File Manager is composed of a main thread and multiple worker threads. When File Manager starts, it reads the configuration file and obtains the required parameters with the `init()` function. The required File Manager parameters are shown in Table 3.1, and the activity diagram of File Manager is shown in Figure 3.2.

The main File Manager thread creates the worker threads. The main thread waits until the Client module sends a request to establish a TCP connection. When that request arrives at the File Manager, the main thread accepts the request and adds the accepted socket descriptor to the epoll buffer. Each worker thread makes a connection to the database and connections to all of the Storage Allocators that are written in the configuration file. The socket descriptors of the Storage Allocators are kept by the `std::map`

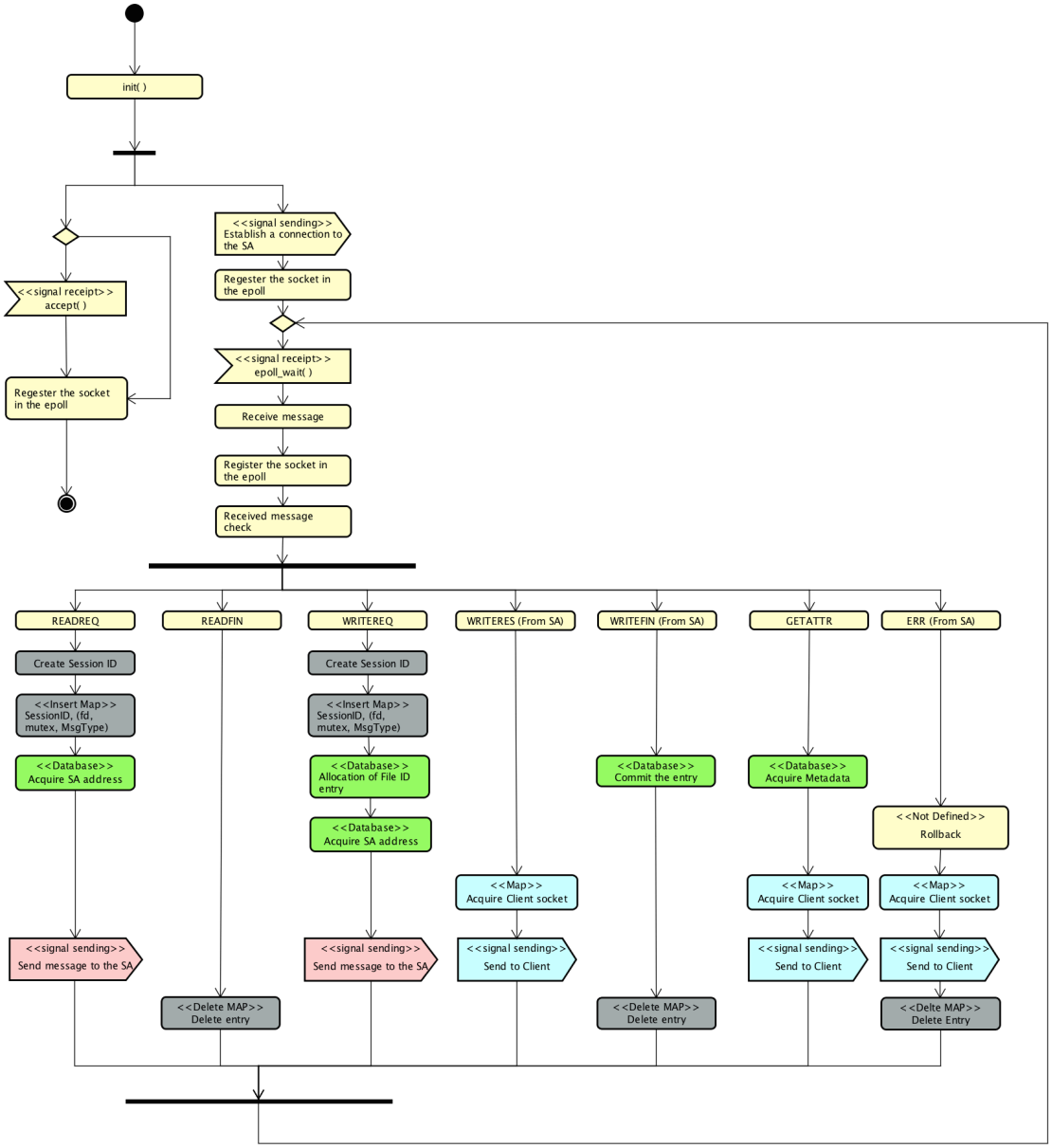


Figure 3.2: Activity diagram of File Manager.

called `SA_MAP`. Then, the worker thread waits using `epoll_wait()` until the message arrives at the socket descriptors that are added to the epoll buffer. When that message arrives, one of the worker threads awakens, receives the message, checks the message type, and calls the function according to the message type.

File Manager accepts three message types from the Client (READ_REQUEST, WRITE_REQUEST, and ATTR_REQUEST) and three message types from the Stor-

age Allocator: READ_FINISH, WRITE_RESPONSE, and WRITE_FINISH. When the message is an ATTR_REQUEST, the worker thread acquires the metadata of the requested file from the database and sends it to the Client module. When the message is a READ_REQUEST, the worker thread creates a Session ID and inserts it and the socket descriptor into the requested Client, the mutex into the socket descriptor, and the message type to the `SESSION_MAP` to store the session information. Then, the worker thread acquires the socket descriptor of the Storage Allocator from the `SA_MAP` and sends the READ_REQUEST on to the Storage Allocator. After the Client module finishes file retrieval, the File Manager receives a READ_FINISH notice from the Client. The worker thread sends the READ_FINISH message to the Storage Allocator and deletes the session information from the `SESSION_MAP`.

When the message is a WRITE_REQUEST, the worker thread creates the Session ID and inserts the session information into the `SESSION_MAP` in the same way as with a READ_REQUEST. Then, the worker thread generates a file ID and allocates a database entry to store file information like file size and FEC parameters. After that, the worker thread acquires the socket descriptor of the Storage Allocator from the `SA_MAP` and sends the WRITE_REQUEST on to the Storage Allocator. Content Espresso is designed so that the Storage Allocator is chosen by the File Manager after taking the required storage service into consideration. In the current implementation, the Storage Allocator is determined by the configuration file. An automatic Storage Allocator selection mechanism is a future goal. After the process of storing the file is completed, the File Manager receives a WRITE_FINISH notice from the Storage Allocator, stores the file information to the database, sends the WRITE_FINISH message to the Client, and deletes the session information from the `SESSION_MAP`.

3.3 Storage Allocator

Storage Allocator is composed of a main thread and multiple worker threads. When Storage Allocator starts, it reads the configuration file and obtains the required parameters with the `init()` function. The required Storage Allocator parameters are shown in Table 3.2, while the Storage Allocator activity diagram is shown in Figure 3.3.

The main thread of the Storage Allocator creates a Chunk Generator accept thread and worker threads. The main thread and the Chunk Generator accept thread wait until a request to establish a TCP connection comes from a File Manager or a Chunk Generator. When the request arrives, the main thread and the Chunk Generator accept thread add the accepted socket descriptor to the epoll buffer. Each worker thread makes a connection to the database and connections to all the cluster heads that are written in the configuration file. Then, the worker thread waits, using `epoll_wait()`, until the message comes to the socket descriptors that are added to the epoll buffer. When the message comes, one of the worker threads awakens, receives the message, checks the message type, and calls the function according to that type.

Storage Allocator accepts three message types from File Manager

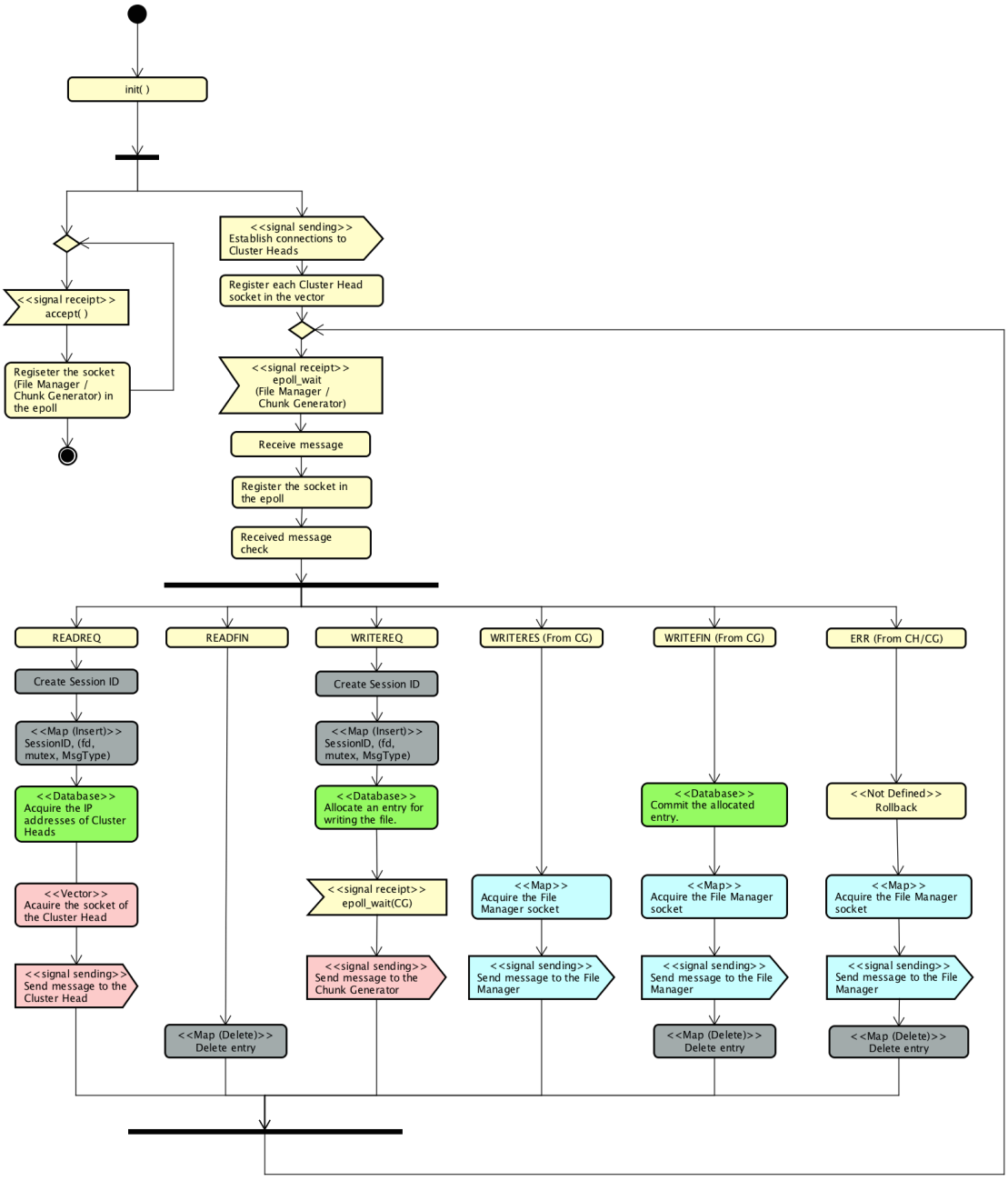


Figure 3.3: Activity diagram of Storage Allocator.

(READ_REQUEST, WRITE_REQUEST, and READ_FINISH) and two message types from Chunk Generator: WRITE_RESPONSE and WRITE_FINISH. When an arriving message is a READ_REQUEST, the worker thread creates a Session ID and inserts it and session information like the socket descriptor into the requested Client,

Table 3.2: Storage Allocator's required parameters.

Type	Name	Description
uint16_t	ch_port	Port number of Cluster Head
uint32_t	cluster_num	Number of Chunk Server Clusters
std::string	ch_addr_0	IP address of Cluster Head 0
std::string	ch_addr_1	IP address of Cluster Head 1
std::string	ch_addr_...	IP address of Cluster Head ...
std::string	ch_addr_n	IP address of Cluster Head n
uint32_t	thread_num	Number of worker threads
uint16_t	cg_accept_port	Port number of Chunk Generator
std::srting	log_file	Path to the log file

the mutex into the socket descriptor, and the message type to the `SESSION_MAP`. Then, the worker thread accesses the database to acquire the Cluster IDs of the Cluster Heads that manage the chunks of the requested file and sends the READ_REQUEST to those Cluster Heads. Content Espresso is designed so that the Cluster Head and Chunk Generator modules are chosen by the Storage Allocator after the required storage service is taken into consideration. In Content Espresso's current implementation, the Cluster Heads and the Chunk Generator are determined by the configuration file. Automated Cluster Head and Chunk Generator selection mechanisms are future goals.

When the arriving message is a WRITE_REQUEST, the worker thread creates a Session ID and inserts the session information into the `SESSION_MAP`, just as occurs with a READ_REQUEST. Then, the worker thread selects the clusters for storing the requested file and allocates the database entry to store the File ID and the Cluster IDs. After that, the worker thread sends the WRITE_REQUEST on to the Chunk Generator. When the arriving message is a WRITE_RESPONSE, the worker thread acquires the socket information from the `SESSION_MAP` and relays it to the File Manager. When the arriving message is a WRITE_FINISH message, the worker thread fixes the allocated database entry from which the previous WRITE_REQUEST came, acquires the socket information from the `SESSION_MAP`, sends the WRITE_FINISH message to the File Manager, and deletes the socket information from the `SESSION_MAP`.

3.4 Chunk Generator

Chunk Generator is composed by a main thread, worker threads, FEC threads, and send threads. When Chunk Generator starts, it reads the configuration file and obtains the parameters with the `init()` function. The Chunk Generator required parameters are shown in Table 3.3, while the Chunk Generator activity diagram is shown in Figure 3.4.

The main thread of the Chunk Generator creates worker threads. The main thread waits until the all the worker threads finish. Each worker thread creates an FEC thread

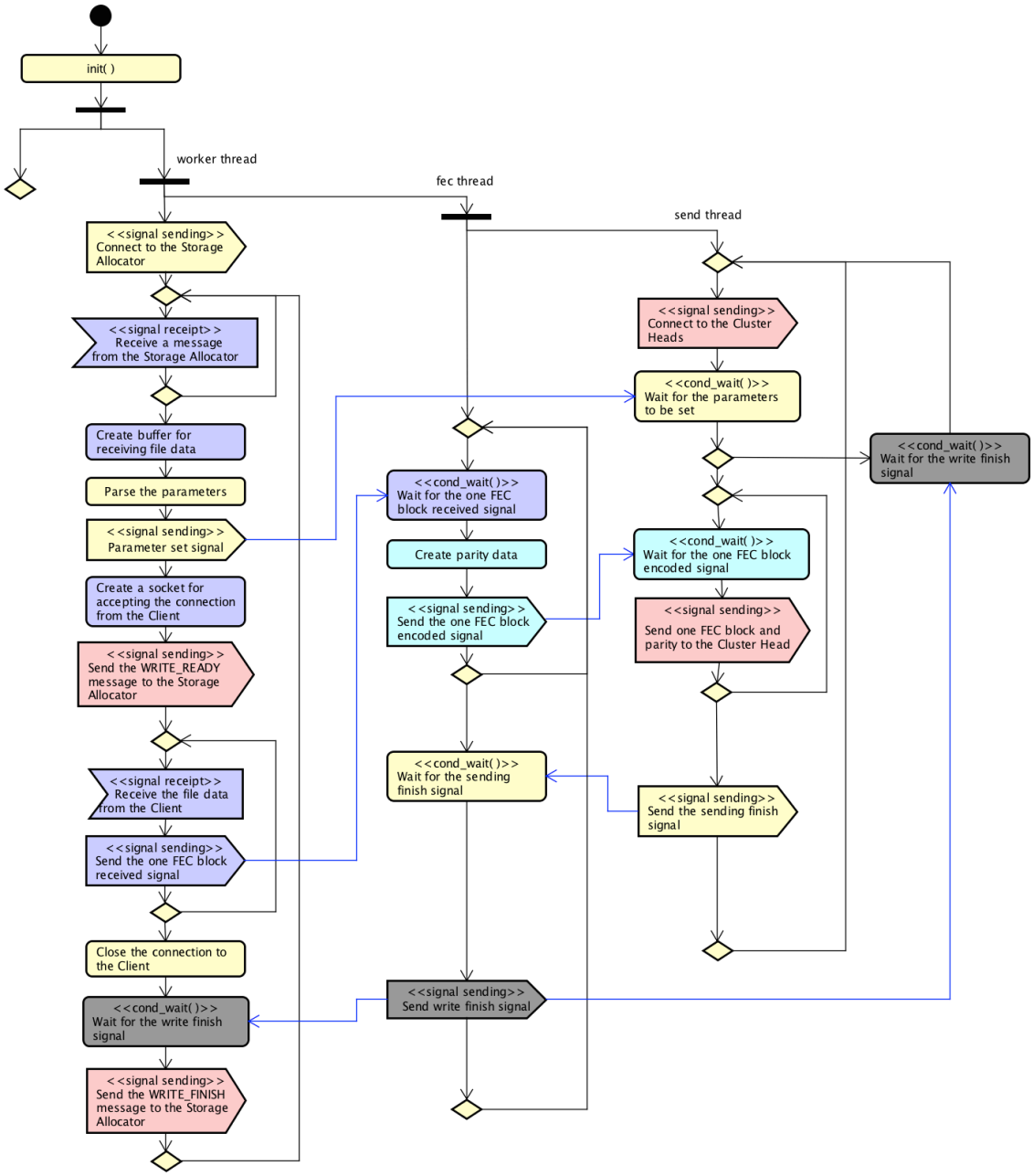


Figure 3.4: Activity diagram of Chunk Generator.

to make parity data for the retrieved file data. Each FEC thread creates a send thread to split the file data and parity data into chunks and send them to the Cluster Heads.

After the worker thread creates the FEC thread, it makes a connection with the Storage Allocator and waits until the WRITE_REQUEST message comes from the Storage Allocator. When it arrives, the worker thread parses the message and creates the buffer

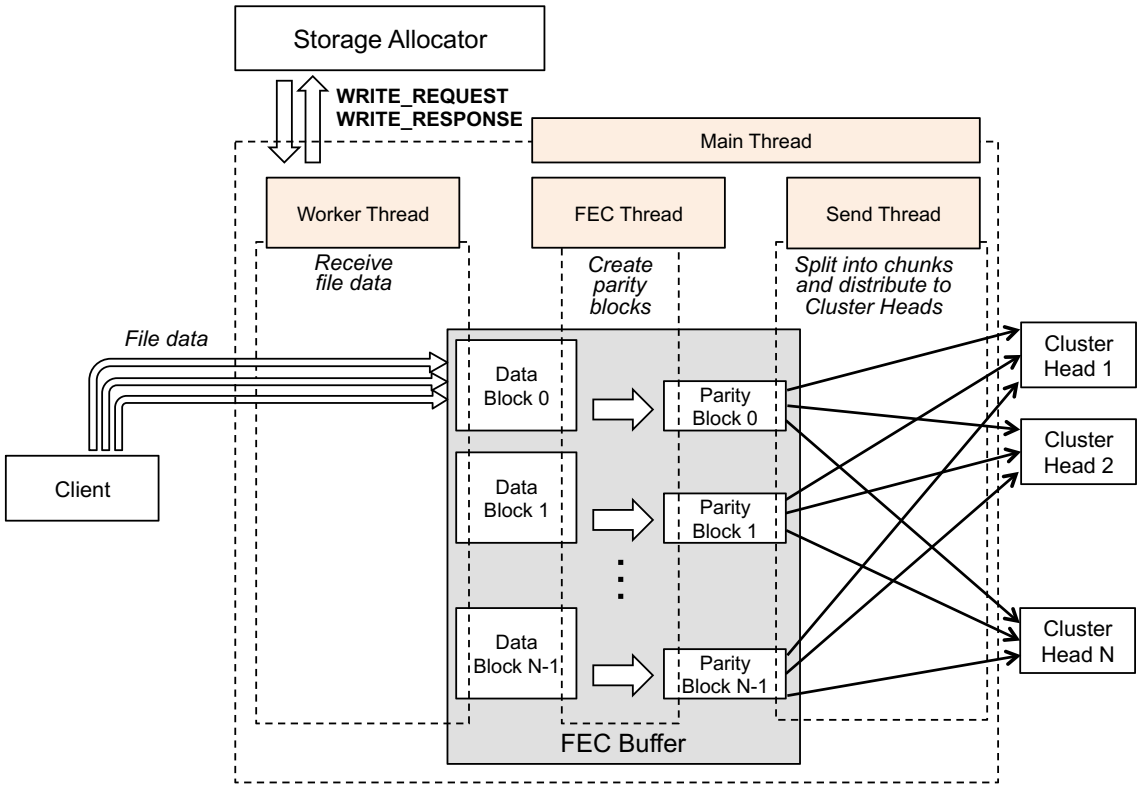


Figure 3.5: A Chunk Generator consists of four types of threads: `Main Thread`, `Worker Thread`, `FEC Thread`, and `Send Thread`, and a single type of buffer (`FEC Buffer`) to multiplex the storing process.

Table 3.3: Chunk Generator's required parameters.

Type	Name	Description
uint16_t	cg_port	Port number of Chunk Generator
std::string	cg_addr	IP address of Chunk Generator
uint32_t	ch_num	Number of Chunk Server Clusters
uint16_t	ch_port	Port number of Cluster Heads
uint16_t	sa_port	Port number of Storage Allocator
std::string	sa_addr	IP address of Storage Allocator
std::string	ch_addr_0	IP address of Cluster Head 0
std::string	ch_addr_1	IP address of Cluster Head 1
std::string	ch_addr_...	IP address of Cluster Head ...
std::string	ch_addr_n	IP address of Cluster Head n
uint32_t	thread_num	Number of worker threads
std::string	log_file	Path to the log file

to receive the file data from the Client. Then, the worker thread informs the send thread of the FEC parameters and Cluster Head information, creates a socket to accept the TCP connection from the Client, and sends the WRITE_RESPONSE message to the Storage Allocator.

When the file data comes from the Client, the worker thread sends a one-FEC block receiving finish signal to the FEC thread for each FEC block occupied by the retrieved file data. Then, the worker thread closes the connection to the Client and waits until the all file storing processes completes before sending the WRITE_FINISH message to the Storage Allocator. The FEC thread waits until the one-block receiving finish signal appears. When the FEC thread receives that signal, it creates parity data for each FEC block. The FEC thread sends one-FEC block encoding finish signal to the send thread for each FEC block encoded, and waits to receive the sending finish signal from the send thread. When that signal arrives, the FEC thread checks that the status of all FEC block indicates that sending has been complete and sends the write finish signal to the worker thread.

The send thread establishes a TCP connection with all the Cluster Heads and waits until the writing parameter comes from the main thread. Then, the send thread parses the received parameters and waits until the one-FEC block encoding finish signal appears, after which the send thread splits the file data and parity data into chunks and sends them to the Cluster Heads. After all the FEC blocks are sent, the send thread sends a sending finished signal to the FEC thread.

3.5 Cluster Head

The Cluster Head is composed of a main thread and multiple worker threads. When the Cluster Head starts, the Cluster Head reads the configuration file and obtains the required parameters with the `init()` function. The required Cluster Head parameters are shown in Table 3.4, while the Cluster Head activity diagram is shown in Figure 3.6.

The main thread of the Cluster Head creates the worker threads. The main thread waits until a request to establish a TCP connection comes from a Storage Allocator or a Chunk Generator, after which the main thread accepts the request and adds the accepted socket descriptor to the epoll buffer. Each worker thread makes a connection to all Chunk Servers that are part of the same cluster and adds the connected socket descriptor to the Chunk Server vector. The Chunk Server vector is implemented by `std::vector` and maintains the connections to the Chunk Servers. Then, the worker thread waits, using `epoll_wait()`, until the message arrives at the socket descriptors that are added to the epoll buffer. When that message arrives, one of the worker threads awakens, receives the message, checks the message type, and calls the appropriate function for the message type.

Cluster Head accepts two message types: READ_REQUESTs from the Storage Allocator and WRITE_REQUESTs from the Chunk Generator. When the arriving message is a READ_REQUEST, the worker thread uses multicast to relay that message

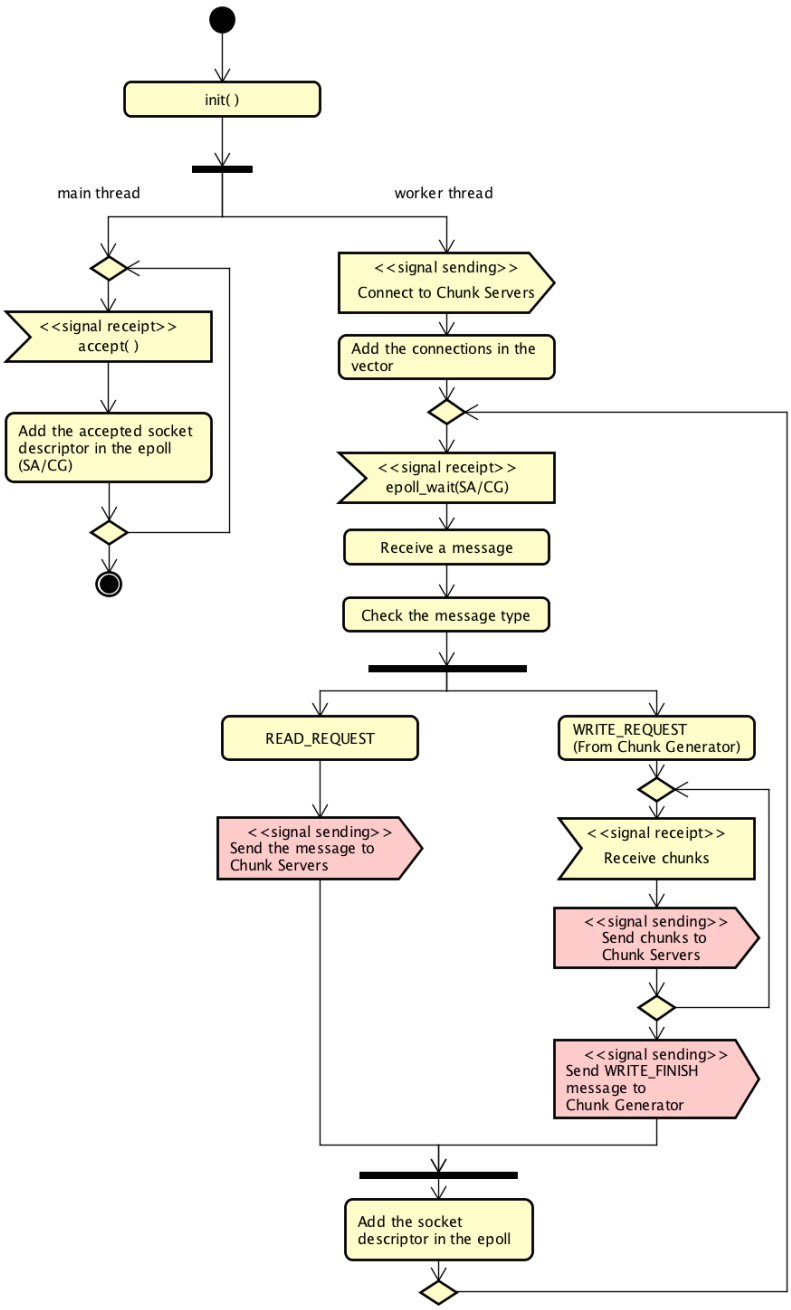


Figure 3.6: Activity diagram of Cluster Head.

to all the Chunk Servers in that cluster. Cluster Head uses multicast so as to reduce delays in chunk sending start time among Chunk Servers. If the arriving message is a WRITE_REQUEST, the worker thread receives the chunks and distributes them to Chunk Servers using TCP and a round-robin algorithm. After sending all the chunks

Table 3.4: Cluster Head's required parameters.

Type	Name	Description
uint16_t	ch_port	Port number of Cluster Heads
uint16_t	cs_udp_port	Port number of Chunk Server (UDP)
uint16_t	cs_tcp_port	Port number of Chunk Server (TCP)
std::string	multicast_family	Multicast address of Chunk Server
uint32_t	cs_num	Number of Chunk Servers
std::string	cs_addr_0	IP address of Chunk Server 0
std::string	cs_addr_1	IP address of Chunk Server 1
std::string	cs_addr_...	IP address of Chunk Server ...
std::string	cs_addr_n	IP address of Chunk Server n
uint32_t	thread_num	Number of worker threads
std::string	log_file	Path to the log file

Table 3.5: Chunk Server's required parameters.

Type	Name	Description
uint16_t	cs_write_port	Port number of Chunk Server (TCP)
uint16_t	cs_read_port	Port number of Chunk Server (UDP)
std::string	multicast_family	Multicast address of Chunk Server
uint32_t	thread_num	Number of worker threads
std::string	log_file	Path to the log file

from the Chunk Generator module, the worker thread sends a WRITE_FINISH message to the Chunk Generator.

3.6 Chunk Server

The Chunk Server is composed of a main thread and multiple worker threads. When the Chunk Server starts, the Chunk Server reads the configuration file and obtains the parameters through the `init()` function. The required Chunk Server parameters are shown in Table 3.1, while the Chunk Server activity diagram is shown in Figure 3.7.

The main thread of the Chunk Server module creates worker threads. The main thread creates a UDP socket to receive the READ_REQUEST and a TCP socket to receive the WRITE_REQUEST from the Cluster Head, then waits until request to establish a TCP connection comes from the Cluster Head. When that request arrives at the Chunk Server, the main thread accepts the request and adds the accepted socket descriptor to the epoll buffer. The worker thread waits, using `epoll_wait()`, until the message comes to the socket descriptors that are added to the epoll buffer. When that message arrives, one of the worker threads awakens, receives the message, checks the message type, and calls

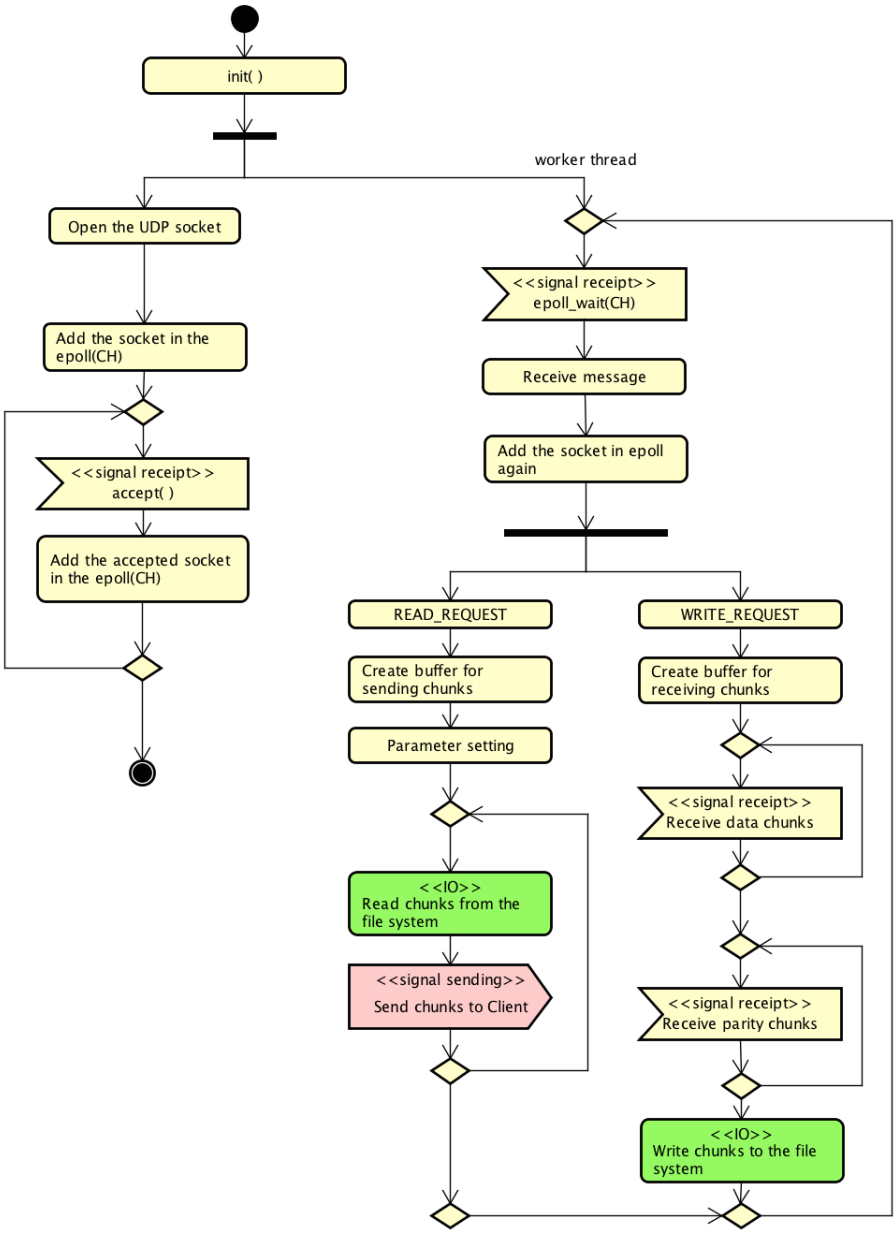


Figure 3.7: Activity diagram of Chunk Server.

the function according to the type of the message.

The Chunk Server accepts two types of messages: READ_REQUESTs and WRITE_REQUESTs from the Cluster Head. When the arriving message is UDP and a READ_REQUEST, the worker thread parses the received message and acquires the file retrieval parameters shown in Table 3.5. Then, the worker thread creates the buffer for reading the chunk data from the local storage system and sets up the file-sending

parameters to the rate control module. The worker thread reads the data chunks from the local file system and sends them to the Client with UDP. When the arriving message is a WRITE_REQUEST, the worker thread creates a buffer for receiving chunks, receives the data chunks and the parity chunks, and stores both the data chunks and the parity chunks in the local files system with the Chunk Server.

3.7 Client

Figure 6.6 shows the implementation details of how the Client receives files. The Client process consists of main thread, recv thread, order thread, and an FEC thread for high throughput chunk receiving and processing. When the Client starts, the main thread of the File Manager creates the recv thread, order thread, and FEC thread. Then, the main thread connects to the Home File Manager and authenticates it.

When the Client sends a file retrieval request to Content Espresso, the main thread of the Client sends the ATTR_REQUEST message including the GFID of the requested file to the File Manager that maintains the file. The GFID includes the File Manager ID, which indicates the File Manager that the Client should send the request. Since Content Espresso does not have a File Manager ID-resolving mechanism in its current implementation, the Client contains the mapping of the File Manager ID and its IP address.

After sending the ATTR_REQUEST, the File Manager returns the file metadata, such as file size and FEC parameters. The main thread allocates Recv Buffer to receive chunks and FEC Buffer to reorder and recover chunks. Then, the main thread sends a file retrieval request to the File Manager. After that, the recv thread receives chunks and stores them in the Recv Buffer. The order thread orders chunks by checking the sequence number inside the chunk headers and storing them in the FEC Buffer. The FEC thread checks each data and parity block and starts recovering unreceived chunks when enough chunks have been received to build original blocks.

When the Client sends a file storage request to Content Espresso, the main thread of the Client sends a WRITE_REQUEST message to the Home File Manager. Then, the main thread receives the assigned GFID, IP address, and port number of the Chunk Generator. After that, the main thread sends the file data to the Chunk Generator. When the entire storage procedure has finished, the main thread receives a WRITE_FINISH message.

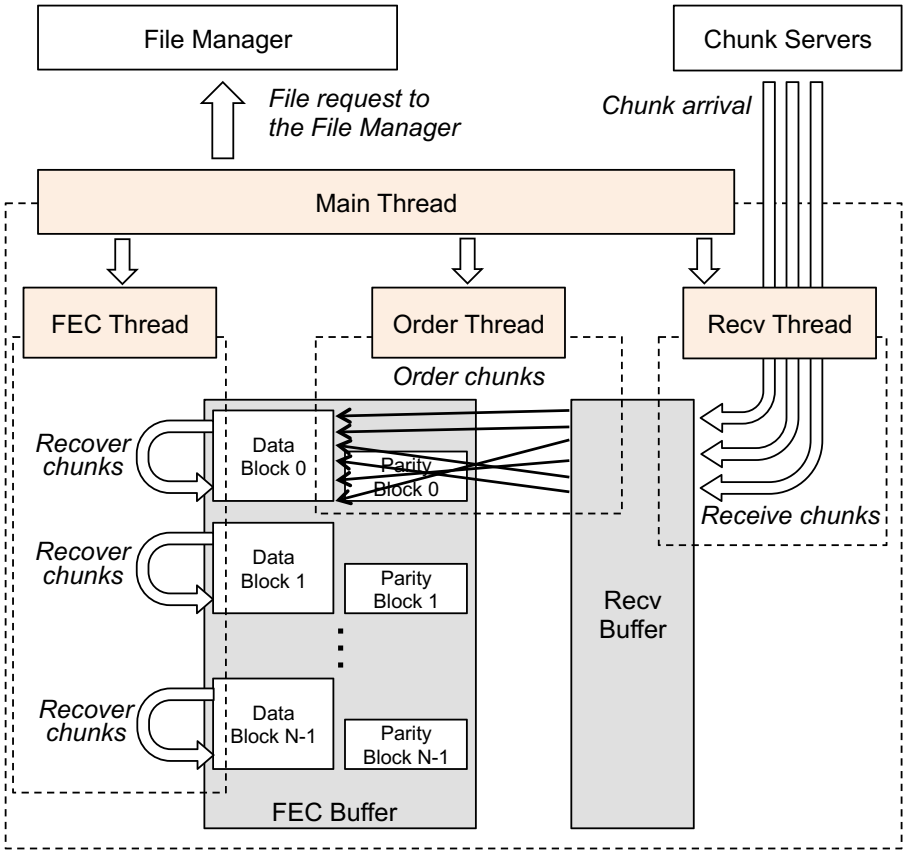


Figure 3.8: A Client consists of four types of threads: `Main Thread`, `Recv Thread`, `Order Thread`, and `FEC Thread`, and two types of buffers (`Recv Buffer`, `FEC Buffer`) to support high throughput retrieval.

Chapter 4

Evaluation of Content Espresso

4.1 Evaluation overview

Content Espresso is designed to achieve large file sharing in a global network environment, with digital content productions used as an example. The previous chapter explained how Content Espresso is designed and implemented to achieve that goal. This chapter evaluates Content Espresso by measuring metadata access performance, file storing performance, and file retrieval performance in an assumed Chunk Server environment by emulating a WAN.

4.2 Evaluation environment

In order to evaluate the performance of the Content Espresso system, 79 physical machines were set up; 72 Chunk Servers, 1 File Manager, 1 Storage Allocator, 1 Chunk Generator, and four Clients. The Chunk Servers, File Manager, and Storage Allocator were all connected to an experimental network with 1Gbps links. The Chunk Generator and Clients were connected with 10Gbps links, because many digital content producers have started to use 10GbE network interface controllers (NICs) as the price of NICs declines. In addition, the capacity of the Internet backbone is increasing; some providers are now providing 10Gbps connectivity to end users. Thus, 10GbE is used in Clients to evaluate the Content Espresso system to take the latest networking realities into consideration. Figure 6.7 shows the experimental environment, while the specifications of each physical machine is shown in Table 6.1.

The 72 Chunk Servers are logically divided into six Chunk Server Clusters by a virtual local area network (VLAN). The Cluster Head module is installed in one Chunk Server machine in each cluster. Three of the four Clients are dummies to emulate simultaneous access to Content Espresso; the remaining Client is used to measure processing time. The average RTT between machines is shown in Table 4.2.

1,272,000-byte, 12,720,000-byte, 127,200,000-byte, and 1,272,000,000-byte files were generated and stored with various FEC block sizes to evaluate the performance

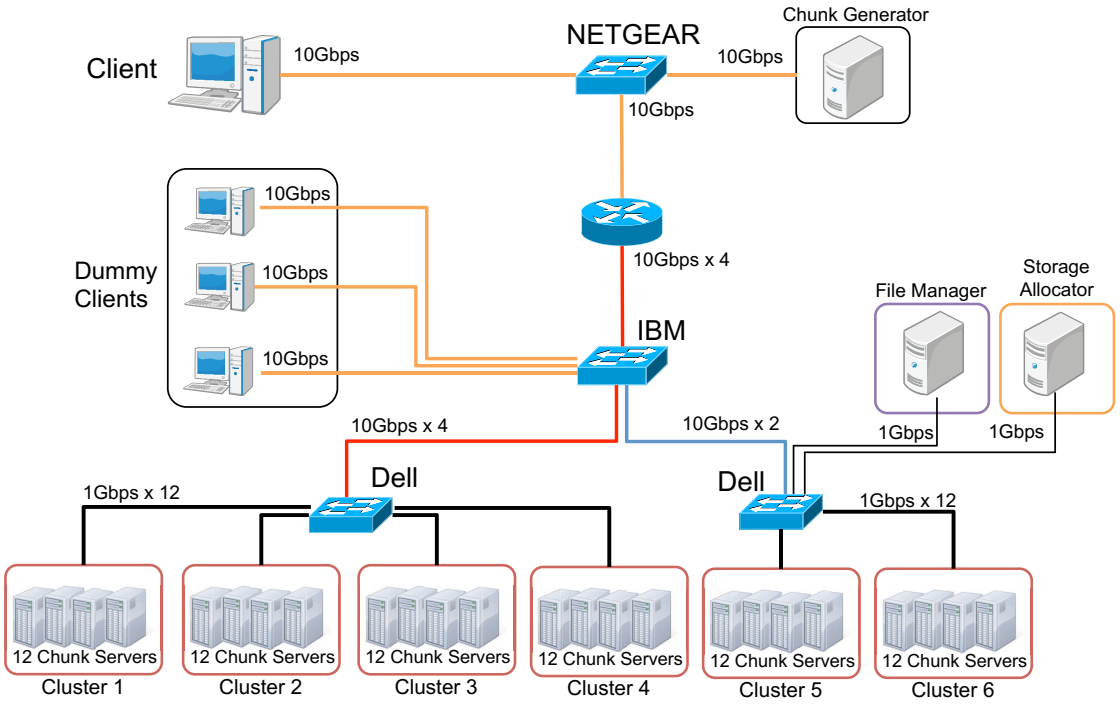


Figure 4.1: Experimental environment for evaluating the Content Espresso system.

Table 4.1: Specifications of physical machines.

Modules	**CPU**	**RAM**	**Network**
Client	Core i7-2600 (3.40GHz)	16GB	10GbE
Dummy Client	Core i7-3770 (3.40GHz)	32GB	10GbE
File Manager	Core i7-3770 (3.40GHz)	8GB	1GbE
Storage Allocator	Core i7-3770 (3.40GHz)	8GB	1GbE
Chunk Generator	Core i7-2600 (3.40GHz)	16GB	10GbE
Chunk Server	Pentium G640 (2.80GHz)	16GB	1GbE

Table 4.2: Average RTTs between machines.

Section			**RTT**
Client	$\Longleftrightarrow$	File Manager	0.36 ms
File Manager	$\Longleftrightarrow$	Storage Allocator	0.23 ms
Storage Allocator	$\Longleftrightarrow$	Chunk Server	0.23 ms
Client	$\Longleftrightarrow$	Chunk Server	0.38 ms
Client	$\Longleftrightarrow$	Chunk Generator	0.40 ms
Storage Allocator	$\Longleftrightarrow$	Chunk Generator	0.27 ms

Table 4.3: Files and FEC block parameters for evaluation.

File Name	Original Size	H_{Data}	# of FEC Block
File A-1	1,272,000 bytes	1,000	1
File B-1	12,720,000 bytes	1,000	10
File B-2	12,720,000 bytes	10,000	1
File C-1	127,200,000 bytes	1,000	100
File C-2	127,200,000 bytes	10,000	10
File C-3	127,200,000 bytes	100,000	1
File D-1	1,272,000,000 bytes	1,000	1,000
File D-2	1,272,000,000 bytes	10,000	100
File D-3	1,272,000,000 bytes	100,000	10

Table 4.4: Additional round-trip network delays between Client and Clusters.

Env.	CL1	CL2	CL3	CL4	CL5	CL6
Delay 0	0ms	0ms	0ms	0ms	0ms	0ms
Delay 1	30ms	30ms	30ms	30ms	30ms	30ms
Delay 2	30ms	30ms	30ms	100ms	100ms	100ms
Delay 3	30ms	30ms	100ms	100ms	200ms	200ms
Delay 4	100ms	100ms	100ms	100ms	100ms	100ms
Delay 5	200ms	200ms	200ms	200ms	200ms	200ms

of Content Espresso system in terms of file size and the number of FEC blocks. Table 4.3 shows the list of files used in the evaluation. H_{Data} is the data block height described in Subsection 2.4.2.

In order to evaluate the file retrieval and storing performance of the Content Espresso system in the assumed network environment, delays and packet loss rates are added to the network with the `tc` command. The File Manager, Storage Allocator, and Chunk Generator are all assumed to be located physically close to the Client in this evaluation, as shown in Figure 4.2. Thus, delay and packet loss is added between the Client and each Chunk Server Cluster (CL1 - CL6).

Table 4.4 shows the combinations of the added round-trip delays. Six patterns (Delay 0 - Delay 5) are used to emulate a global network environment. The round-trip delay times of 30ms, 100ms, and 200ms were chosen to reflect actual network delays. Assuming that the machine with the Client module is in Japan, 30ms, 100ms, and 200ms delays suggest that the Chunk Server Clusters are located in Japan, the US, and Europe respectively. Table 4.5 shows the combinations of the added packet loss rates. Seven patterns (Loss 0 - Loss 6) were used to emulate a lossy network environment. The packet loss rate of 0.3% was chosen by to reflect actual service level agreements (SLAs) in ISPs [13, 14].

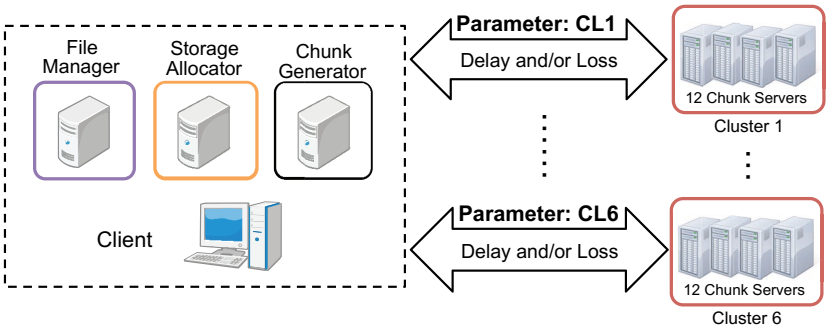


Figure 4.2: Emulated global environment for evaluating the Content Espresso system.

Table 4.5: Additional network packet loss rates between Client and Clusters.

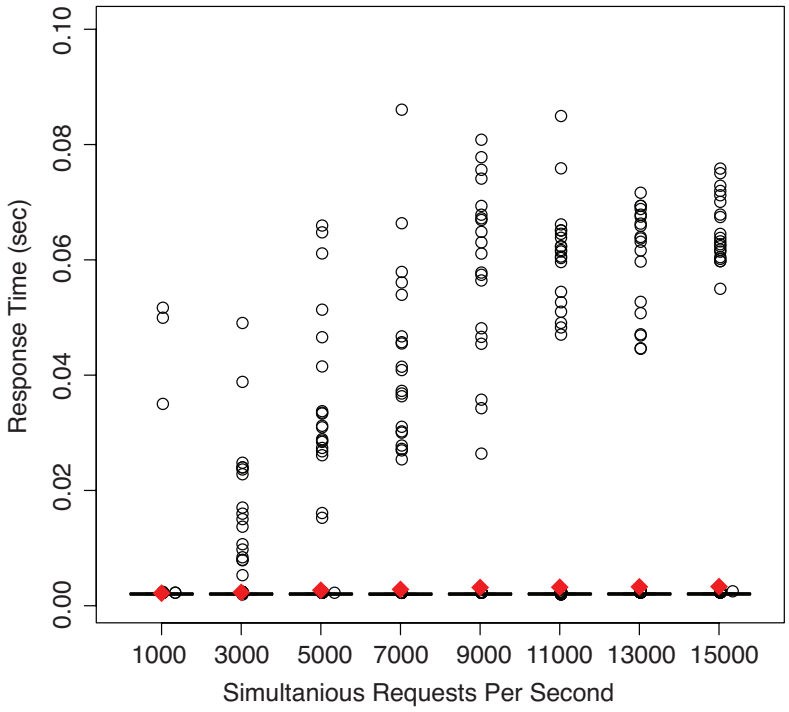
Env.	CL1	CL2	CL3	CL4	CL5	CL6
Loss 0	0%	0%	0%	0%	0%	0%
Loss 1	0.3%	0%	0%	0%	0%	0%
Loss 2	0.3%	0.3%	0%	0%	0%	0%
Loss 3	0.3%	0.3%	0.3%	0%	0%	0%
Loss 4	0.3%	0.3%	0.3%	0.3%	0%	0%
Loss 5	0.3%	0.3%	0.3%	0.3%	0.3%	0%
Loss 6	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%

4.3 System performance

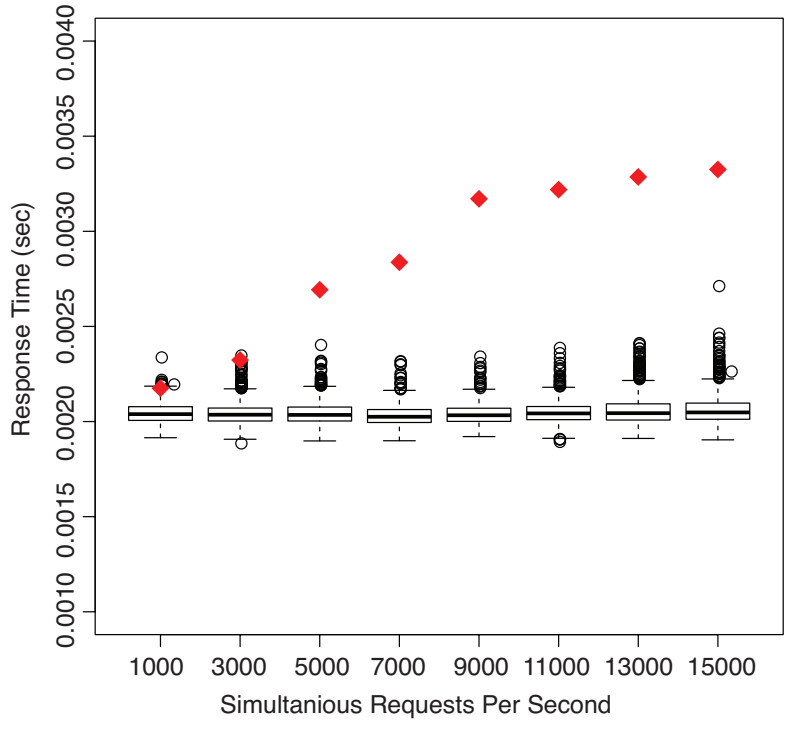
4.3.1 Metadata access performance

Although Content Espresso distributes the metadata of stored files to File Managers, multiple processes requesting access to the same file simultaneously causes a high workload for the File Manager and may lead to performance degradation. In order to evaluate the File Manager's simultaneous processing capability, the metadata request response time of the Client was measured; three dummy clients sent metadata requests to the File Manager ranging from 1,000 requests per second to 15,000 requests per second.

Figure 4.3 shows the box plots of the results: the lower graph is an enlarged version of the upper graph. The diamond-shaped dots indicate the average value, while the circles indicate outliers. Although the average response time gradually increases as the number of simultaneous requests increases, the median does not increase until 15,000 requests per second is reached. The metadata request response time was not measured above 15,000 requests per second because there is the bottleneck of the number of connections in File Manager. These results thus confirm that a single File Manager can deal with fewer than 15,000 simultaneous requests per second without performance degradation, which is sufficient from the viewpoint of the C10K problem [15].



(a) Y-axis is between 0 to 0.1.



(b) Y-axis is between 0.001 to 0.004.

Figure 4.3: Metadata response time of the File Manager when simultaneous requests arrive.

4.3.2 File retrieval performance

When the Content Espresso system receives a file retrieval request through a Client module, the Chunk Servers send chunks to the Client using UDP at the requested rate. When that rate is 1,000Mbps and there are 72 Chunk Servers, each Chunk Server sends chunks to the Client at $1,000/72 \approx 13.9$ Mbps. After the Client has received the chunks to a certain extent (typically, the H_{Data} chunks have arrived), it tries to recover non-received chunks until all chunks are recovered. In this evaluation, the period of time between the moment when the Client requests file retrieval from the File Manager and the moment when the Client finishes receiving and recovering all chunks was measured to reveal the effects of file size and the location of Chunk Server Clusters on file retrieval performance.

File retrieval throughput in various file sizes

In order to evaluate the file retrieval performance at various file sizes, the file retrieval time and file retrieval throughput were measured using Files A-1, B-2, C-3, and D-3 (Table 4.3) by changing the requested file retrieval rate from 500Mbps to 3.5Gbps. Figure 4.4 lays out the relationship between the requested file retrieval rate and actual file retrieval throughput. The actual file retrieval throughput of File D-3 is almost same as the requested file retrieval rate, while the actual file retrieval throughput of File A-1 is no higher, even though the requested file retrieval rate is much higher. The performance degradation for small-size files is caused by the overhead of accessing File Manager before chunk retrieval. Therefore, the larger a file is, the better the file retrieval performance in the Content Espresso system.

This evaluation also reveals that the maximum file retrieval rate in Content Espresso is around 3.0 - 3.5Gbps. Most files cannot be recovered by FEC when the rate is over 3.5Gbps because of overflow in the UDP socket buffer that is due to the Linux kernel's performance problems in UDP retrieval. High-speed packet IO libraries such as netmap [16] and DPDK [17] could help solve this problem.

File retrieval time comparison in various network environments of Chunk Server Clusters

In order to evaluate the file retrieval performance in various network environments of Chunk Server Cluster locations, file retrieval times with Content Espresso and TCP-based chunk retrieval were measured by changing the Chunk Server Cluster network environment (Tables 4.4 and 4.5). TCP-based chunk retrieval is implemented to compare the file retrieval performance of Content Espresso to that of existing TCP-based chunk retrieval. In TCP-based chunk retrieval, the Client establishes a TCP connection with each Chunk Server and retrieves data chunks of the target file. In Content Espresso, the file retrieval time was measured at file retrieval rates of 1Gbps, 2Gbps, and 3Gbps.

Figures 4.5 - 4.7 present the comparison of the file retrieve time when the Chunk Server Clusters are located in a network environment with additional levels of delay.

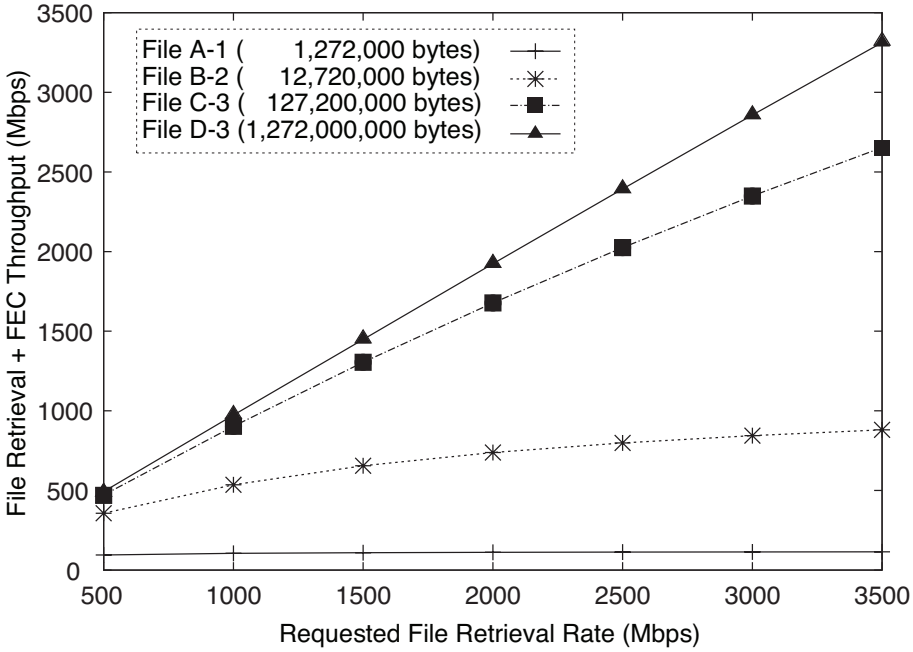


Figure 4.4: File retrieval throughput using A-1, B-2, C-3, and D-3 files.

TCP shows better performance than Content Espresso (3Gbps) in Delay 0, which emulates an environment in which all Chunk Servers are located in the Client's LAN, regardless of file size. TCP can utilize a 10Gbps link more efficiently than the Content Espresso system when the RTT and packet loss is small because the maximum file retrieval performance of the Content Espresso system is about 3.0 - 3.5Gbps in its current implementation. However, Delay 0 is not a realistic environment because the Content Espresso system assumes that Chunk Server Clusters are installed in multiple organizations.

TCP has also better performance than Content Espresso (3Gbps) in Delay 1 and Delay 2 when File D-1 is used. In general, TCP throughput is initially low and becomes higher because of TCP's slow start. Since the file retrieval is finished before the TCP window size becomes large enough to make a difference when a file is small, its average file retrieval performance is low. Thus, TCP performance for File D-1 is better than for File B-1 or File C-1. The request overhead of the Content Espresso system becomes larger when the RTT between the Client and the Chunk Servers is longer because the Content Espresso system uses TCP to send the request messages to the Chunk Servers. However, the chunk retrieval time of the Content Espresso system is not affected by the RTT. Since the request messages are quite small in comparison to the chunk data, the request overhead can be ignored with larger file sizes. Therefore, it is not necessary for the Content Espresso system to consider and adjust for the locations of the Chunk Servers.

Figures 4.8 - 4.10 present the comparative file retrieval times when Chunk Server Clusters are located in a lossy network environment. In this experiment, a 30ms RTT for each Chunk Server Cluster is added in the same environment as Delay 1 to emulate a realistic environment. The Content Espresso system (3Gbps) has better performance than TCP, except that File D-1 is used in the Loss 0 environment because of TCP's slow start and the performance limits of Content Espresso described above. The request overhead of the Content Espresso system is not affected by packet loss but by the RTT between the Client and the Chunk Servers. While TCP is significantly affected by packet loss, the chunk retrieval time of the Content Espresso system is not affected. Therefore, the Content Espresso system is tolerant of packet loss, and it is not necessary to consider and adjust for the network conditions of the Chunk Server Clusters, as long as the network provider ensures that the packet loss rate is within SLA parameters.

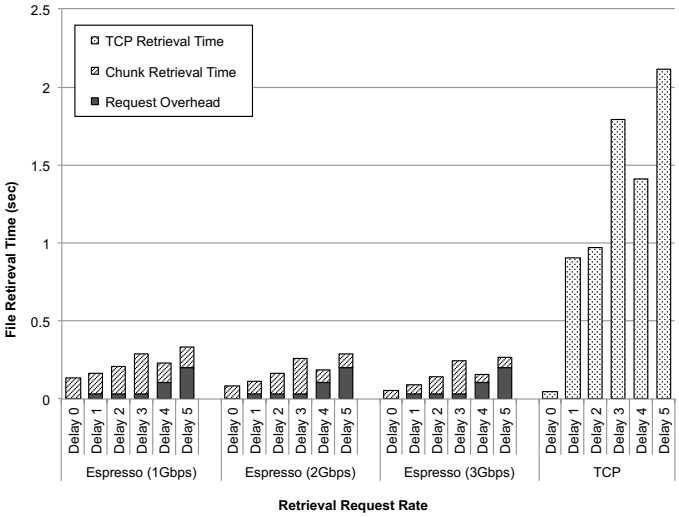


Figure 4.5: File B-1 retrieval time under various delay environments.

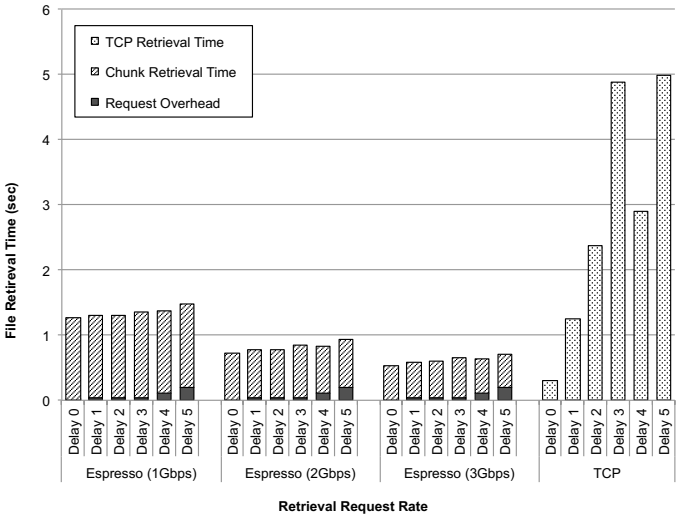


Figure 4.6: File C-1 retrieval time under various delay environments.

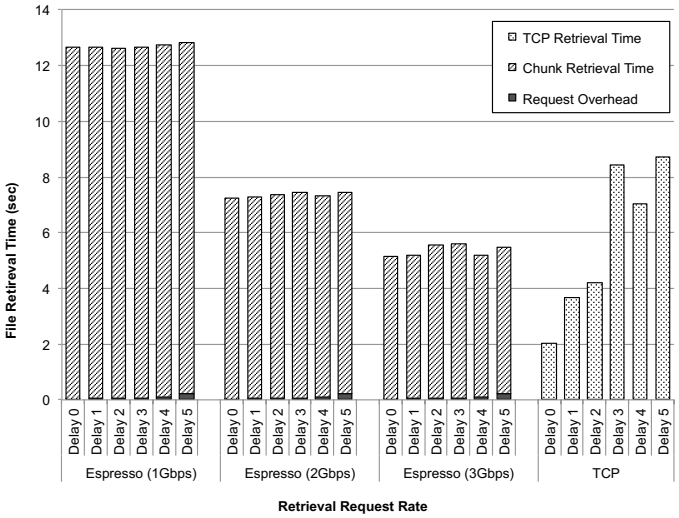


Figure 4.7: File D-1 retrieval time under various delay environments.

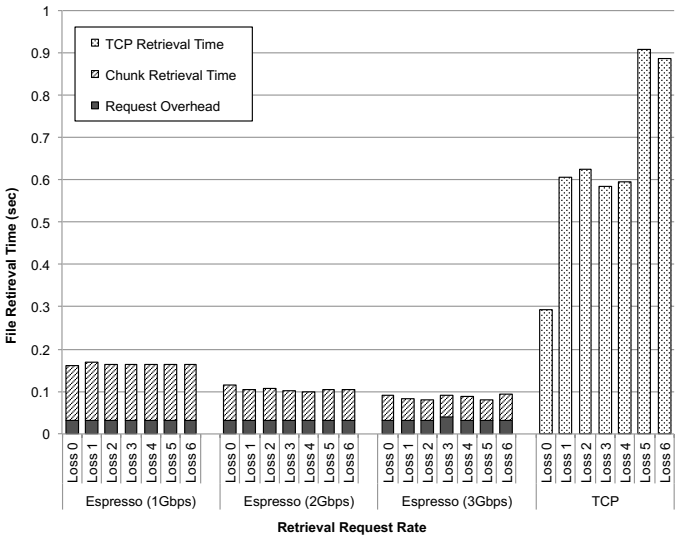


Figure 4.8: File B-1 retrieval time under various loss environments.

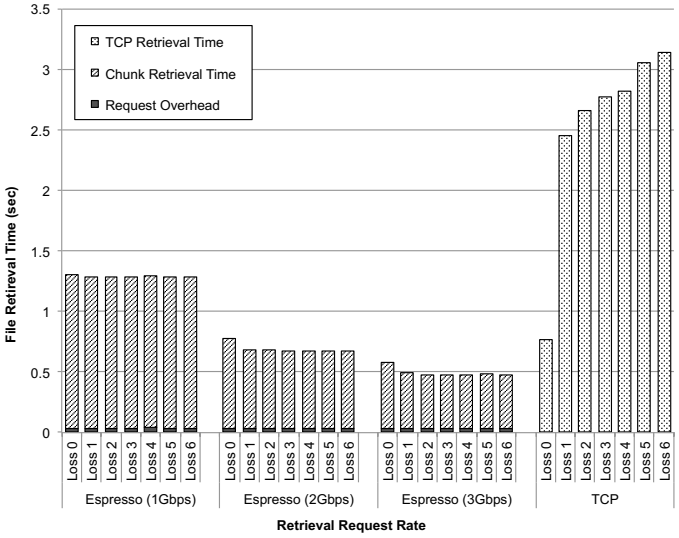


Figure 4.9: File C-1 retrieval time under various loss environments.

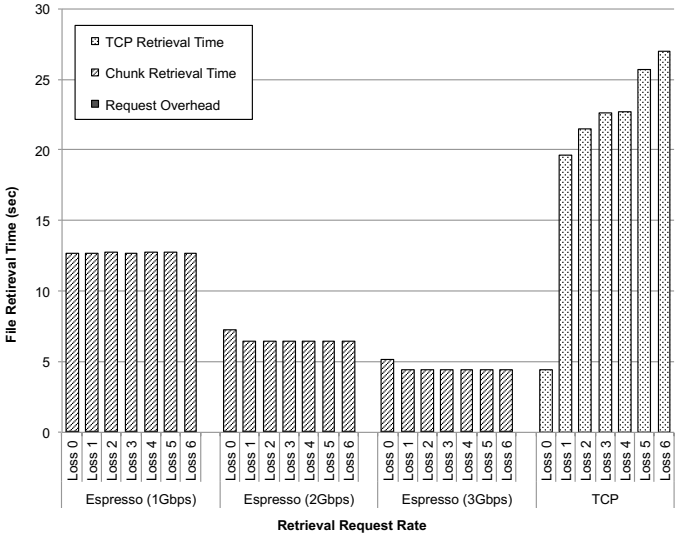


Figure 4.10: File D-1 retrieval time under various loss environments.

4.3.3 File storing performance

When a Client stores a file to the Content Espresso system, it sends the file data to the Chunk Generator using TCP. The Chunk Generator has multiple threads that can simultaneously process different pieces of the file data. Thus, after the Chunk Generator receives part of the file data for generating a single FEC block from the Client, it starts making parity data of the file data, splitting the file data and parity data into chunks, and sending them to the cluster heads one after another. In this evaluation, the period of time between starting to send the file to the Chunk Generator and receiving the WRITE_FINISH message was measured.

File storing throughput in various file sizes

In order to evaluate file storing performance at various file sizes, the file storing time was measured and file retrieval throughput calculated using the files shown in Table 4.3. Figure 4.11 describes the file storing throughput. The result shows that larger files have better performance than smaller files, which is caused by the overhead of establishing a TCP connection between the Client and the Chunk Generator. Establishing a TCP connection before beginning to store a file might improve this performance. However, this is not a practical solution because the Content Espresso system assumes that chunk generators are chosen dynamically for each request, after taking into account the distance from the Client and its workload.

The results also show that, with the same file size, a small FEC block has better performance than a large FEC block, because the Chunk Generator can start generating parity data and sending chunks to the cluster heads as soon as the Chunk Generator receives file data that is the size of the FEC block. Therefore, using large files and small FEC blocks is key to high throughput file storing. It would be possible to store a large number of small files with high throughput by interleaving file storing, e.g., by sending several requests simultaneously.

File storing time comparison in various network environments of Chunk Server Clusters

In order to evaluate file storing performance in different network environments and Chunk Server Cluster locations, file storing times were measured as network Chunk Server Clusters environment changed as shown in Tables 4.4 and 4.5.

Figure 4.12 presents the file storing time of File C-1 when the Chunk Server Clusters are located in a network environment with an additional delay. The results show that file storing times in the Delay 3 and Delay 5 environments are slower than in the other environments, which is due to the fact that Delay 3 and Delay 5 both contain 200ms worth of extra distance from the Chunk Server Clusters. Figure 4.13 presents the file storing time of File C-1 when the Chunk Server Clusters are located in a lossy network environment. In this experiment, 30ms of RTT is added to each Chunk Server Cluster in the same environment as Delay 1 to emulate a practical environment. The result

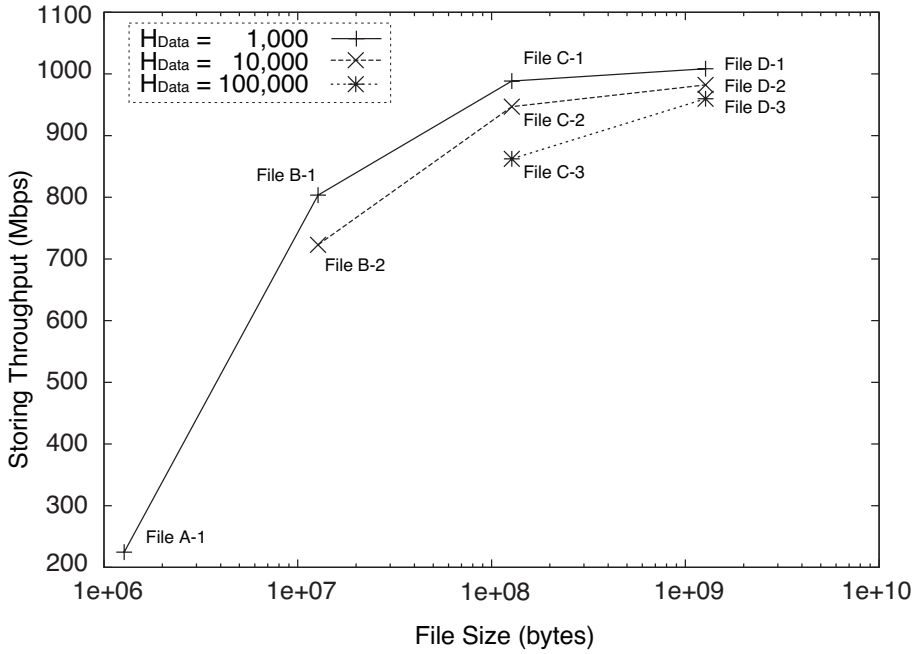


Figure 4.11: File storing throughput using files shown in Table 4.3.

shows that any lossy network environments (Loss 1 - 5) cause significant performance degradation of file storing.

These results confirm that the file storing performance of the Content Espresso system becomes worse if even only one of the Chunk Server Clusters is located far away from the Client or is in a lossy network environment. However, the Content Espresso system focuses on fast file retrieval because it is designed for large file sharing among digital content producers, who operate in a sector in which large and even massive files are stored once and retrieved many times. Thus, the performance degradation of the file storing performance does not significantly affect the overall performance of Content Espresso; the same conclusion can be drawn when File B-1 and D-1 are used for this particular evaluation.

4.4 Appropriate FEC block size

Content Espresso allows users to specify the FEC block size for each file. Using large-size FEC blocks causes performance degradation because of the amount of time it takes to generate parity data and to recover lost chunks. On the other hand, small-size FEC blocks are weak in terms of burst packet loss and chunk recovery failure. For this part of the evaluation of Content Espresso, the file retrieval success rate was measured using different retrieval rates and various file types (shown in Table 4.3) to reveal the most appropriate FEC block size.

Figure 4.14 presents the retrieval success rate as changing the file retrieval rate using

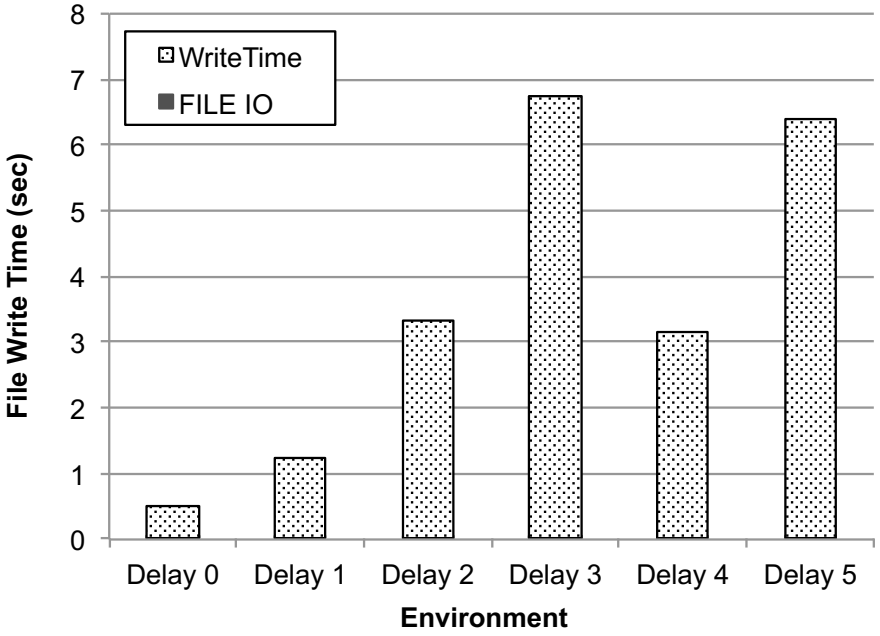


Figure 4.12: File storing throughput under various delay environments using the files shown in Table 4.3.

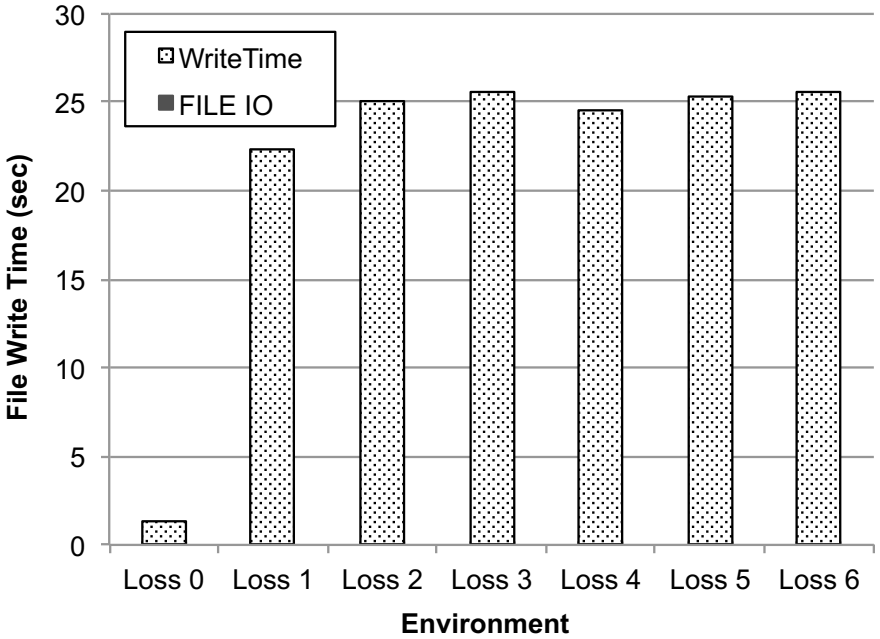


Figure 4.13: File storing throughput using the files shown in Table 4.3.

a 1,272,000,000-byte file when H_{Data} is 1,000, 10,000, or 100,000. The number of FEC

blocks is 1, 10, and 100 when H_{Data} is 100,000, 10,000, and 1,000 respectively. The results show that the retrieval success rate is more than 80% when H_{Data} is 10,000 and 100,000, but that it decreases rapidly when H_{Data} is 1,000.

Figure 4.15 presents the retrieval success rate as changing the file retrieval rate when H_{Data} is 1,000 and 1,272,000-byte, 12,720,000-byte, 127,200,000-byte, and 1,272,000,000-byte files are utilized. The number of the FEC blocks is 1, 10, 100, and 1,000 when the file size is 1,272,000 bytes, 12,720,000 bytes, 127,200,000 bytes, and 1,272,000,000 bytes respectively. The results show that the retrieval success rates of the 1,272,000-byte file and the 12,720,000-byte file are above 80% when the retrieval rate is between 500Mbps and 3,000Mbps.

Taken together, these results confirm that the Content Espresso system shows better file retrieval success rates when the number of FEC blocks is 1 or 10. Therefore, this evaluation concludes that the appropriate FEC block size is approximately 10% of the original file size.

4.5 System availability

Content Espresso utilizes multiple commodity servers as Chunk Servers, which must have lower availability than special servers designed for providing high throughput and high-availability storage and retrieval services. The recovery capability of the Content Espresso system is determined by the redundancy rate (H_{Parity}/H_{Data}). When the redundancy rate is r, $H_{Data} * r/2$ of lost chunks can be recovered by the LDGM coding in the Client. For example, when H_{Data} is 1,000 and H_{Parity} is 200, 100 chunks lost in each block can be recovered. The total availability A of the Content Espresso system can be calculated as follows, when N is the total number of Chunk Servers, n is the number of tolerable Chunk Server failures, and a is the availability of each Chunk Server:

$$A = \sum_{k=0}^{n} (1-a)^k a^{N-k} {}_n\mathrm{C}_k$$

In this evaluation, the total availability of the Content Espresso A is calculated when a user appends 10%, 20%, or 30% redundant data to the original file. Then, the relationship among the availability of each Chunk Server, the number of Chunk Servers, and the availability of Content Espresso and its conditions for achieving availability at the five nines (99.999%) level is discussed.

Figure 4.16 lays out the relationship between the number of Chunk Servers and the total availability when the redundancy rate is 10%. The file availability decreases as the number Chunk Servers increases when the availability of each Chunk Server is less than 0.95. To achieve five nines (99.999%) availability, the system requires at least 200 Chunk Servers with 0.99 availability.

Figure 4.17 presents the relationship between the number of chunk Servers and the total availability when the redundancy rate is 20%. The file availability increases as

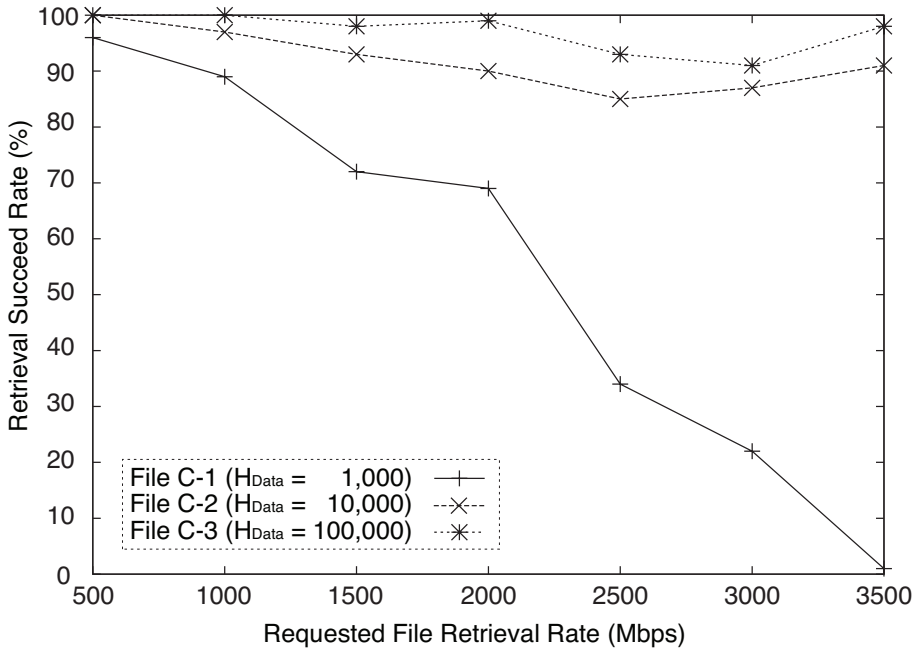


Figure 4.14: File retrieval and error recovery success rates for a 127,200,000-byte file.

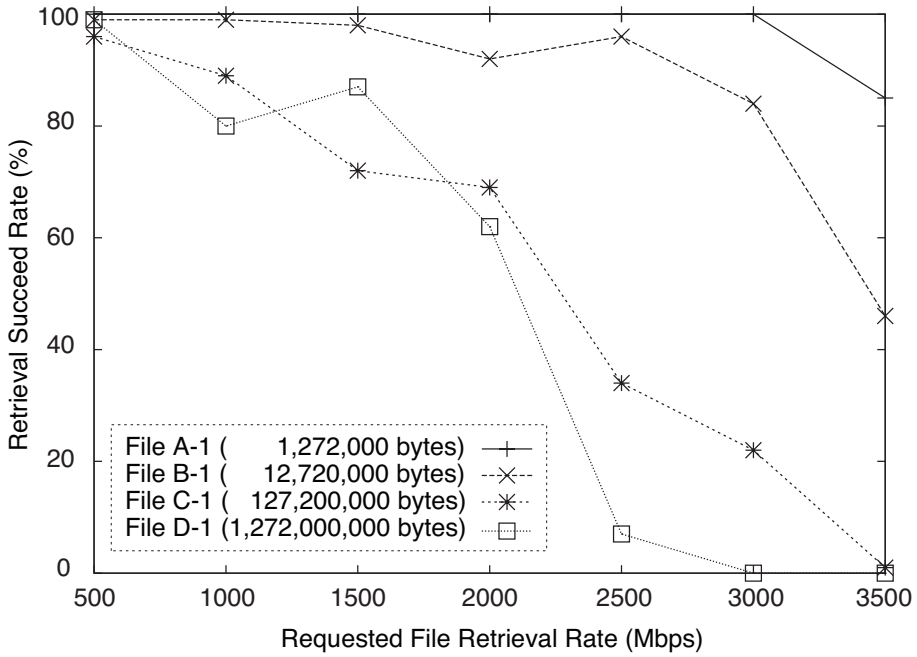


Figure 4.15: File retrieval and error recovery success rates when $H_{Data} = 1,000$.

the number of Chunk Servers increases when the availability of each Chunk Server is more than 0.91. To achieve five nines (99.999%) availability, the system requires 50 Chunk Servers with 0.99 availability, 60 Chunk Servers that have 0.99 availability, 100 Chunk Servers with 0.98 availability, 160 Chunk Servers that have 0.97 availability, or 240 Chunk Servers that have 0.96 availability.

Figure 4.18 shows the relationship between the number of Chunk Servers and the total availability when the redundancy rate is 30%. The file availability increases as the number of Chunk Servers increases at any kind of Chunk Server availability. To achieve five nines (99.999%) availability, the system requires 40 Chunk Servers that have 0.99 or 0.98 availability, 60 Chunk Servers that have 0.97 availability, 100 Chunk Servers that have 0.96 availability, 120 Chunk Servers that have 0.95 availability, 160 Chunk Servers that have 0.94 availability, or 220 Chunk Servers that have 0.93 availability.

4.6 Summary

This section evaluated Content Espresso by measuring metadata access performance, file retrieval performance, and file storing performance in an assumed Chunk Server environment by emulating a WAN. Then, the appropriate FEC size and the system availability of Content Espresso were discussed. The result of the metadata access performance evaluation confirms that Content Espresso can deal with more than 15,000 simultaneous requests per second to a single File Manager without performance degradation. The result of the file retrieval performance evaluation confirms that Content Espresso can deliver stored files faster than TCP-based protocol when the RTT between the Chunk Servers and the Client is longer than 30ms or only one of the Chunk Server Clusters is in a lossy network environment. The result of the file storing performance revealed that the file storing performance becomes worse if even only one of the Chunk Server Clusters is located far away from the Client or is in a lossy network environment. The evaluation of the FEC block size showed that Content Espresso should chose the approximately 10% of the original file size as a FEC block size. Finally, the evaluation of the system availability shows the relationship between the number of Chunk Servers and the total availability. Users can select neccesary redundancy rate by using the results.

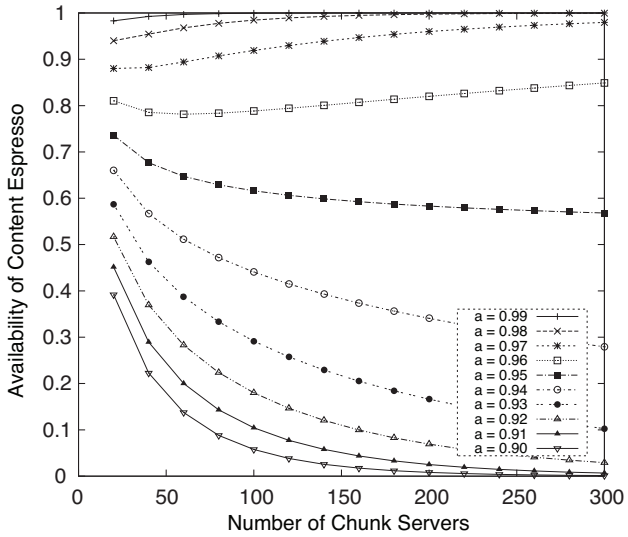


Figure 4.16: Relationship between file availability and the number of Chunk Servers when 10% redundancy is appended.

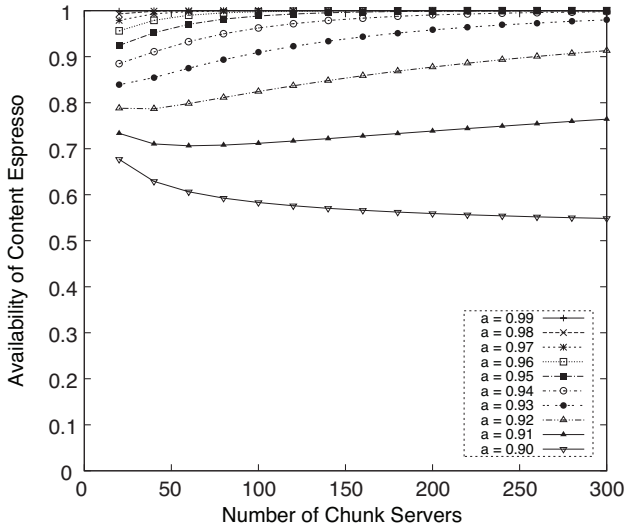


Figure 4.17: Relationship between file availability and the number of Chunk Servers when 20% redundancy is appended.

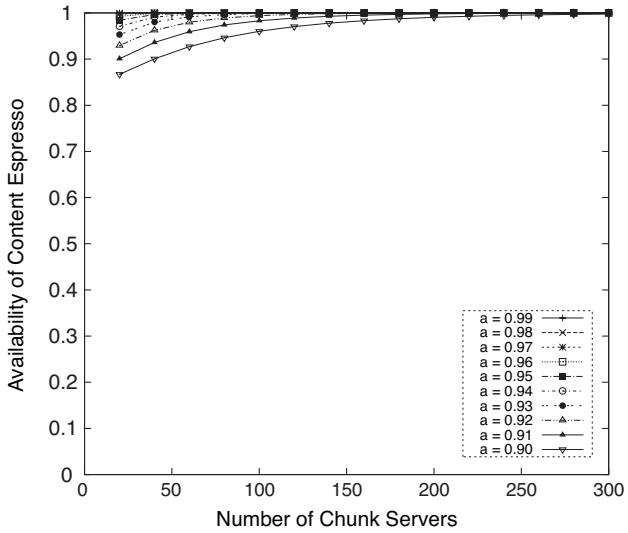


Figure 4.18: Relationship between file availability and the number of Chunk Servers when 30% redundancy is appended.

Chapter 5

Demitasse: A Network-Oriented UHD Video Playback System

5.1 Background

With the decreasing price of equipment for video productions, many video production companies have started making high-resolution content in UHD (3840 x 2160 pixels). Generally, a video producer shoots source videos and stores them as a sequence of video component files. For example, the first component file is named *FILE_001*, the second *FILE_002*, etc. For low cost and quick video creation, video producers share content files with post-production companies all over the world using shared storage that is connected to the Internet. Post-production companies must retrieve these content files from storage and process them according to the instructions of the video producer. Post-production companies must be to play video back easily, but this can be difficult. First, a post-production company has to retrieve the component files of a particular video production from the shared storage. Then, they have to play the component files back, usually in order, with a video playback application. This remains challenging even when Internet access to the storage servers is as rapid as 10Gbps.

This chapter proposes *Demitasse*, a network-oriented video playback system using Content Espresso. The performance requirements of Demitasse are as follows: 1) Demitasse must play uncompressed UHD video stored in shared storage on the Internet; 2) Demitasse must be able to edit the order of component files and share them immediately without retrieving all the component files; 3) Demitasse must not require making duplicate copies of component files. In order to satisfy these requirements Demitasse uses two technologies: Content Espresso and *Catalogue System*. For storing video component files, Demitasse uses Content Espresso, a distributed storage system for large file sharing. Content Espresso enables low cost data storage and high throughput file transmission, regardless of the location of clients, by using FEC and UDP. To describe and store relationships among component files, Demitasse uses Catalogue System, a distributed graph database. Catalogue System enables users to edit file relations by using a

directed graph. In this chapter, Demitasse is designed, implemented, and its performance evaluated. The results confirm that Demitasse achieves uncompressed UHD playback at 30fps.

5.2 Catalogue System

5.2.1 Cencept of the Catalogue System

With the rapid growth of content files on the Internet, text-based search engines such as Yahoo! and especially Google have been developed and are widely used to search for specific content. However, one problem with text-based search engines is that they require knowing the appropriate keywords beforehand, even when those who want to find something do not have deep insights into the topic. Catalogue System stores and shares catalogues globally in a distributed and autonomous way. A catalogue represents at least one relation among files by a directed graph. An object is defined as either a file or a catalogue. A catalogue can associate not only files but also other catalogues with one another. Looking up catalogues from a focused catalogue is also possible. The size of a catalogue is not restricted. Figure 5.1 represents an overview of objects and three kinds of users called *File Owners*, *Cataloguers*, and *Clients*. File owners create files and make them shareable over the Internet. Each shared file is managed by the file owner and is identified by a globally unique identifier. A cataloguer creates catalogues by choosing one or more pairs of objects and make those catalogues shareable on the Internet. Clients retrieve catalogues containing a given object with a reverse lookup approach. They can navigate across directed graphs in the catalogues to reach another object.

5.2.2 System architecture

The Catalogue System is composed of three modules: File Manager, Catalogue System, and Graph Manager.

File Manager

The File Manager manages files identified by their GFIDs (as a reminder, global file IDs). A Single File Manager is located in the domain of a file owner. When a client requests a file specified by a GFID to the File Manager identified by the File Manager ID in the GFID, the File Manager returns that file to the client.

Catalogue Server

The Catalogue Server manages the catalogues. It is located in the domain of a cataloguer. Once a cataloguer obtains a Catalogue Server ID, a Catalogue Server can be launched. The cataloguer can store catalogues and control the access to each of their catalogues. The Catalogue Servers are managed in a distributed manner so that the cataloguers are

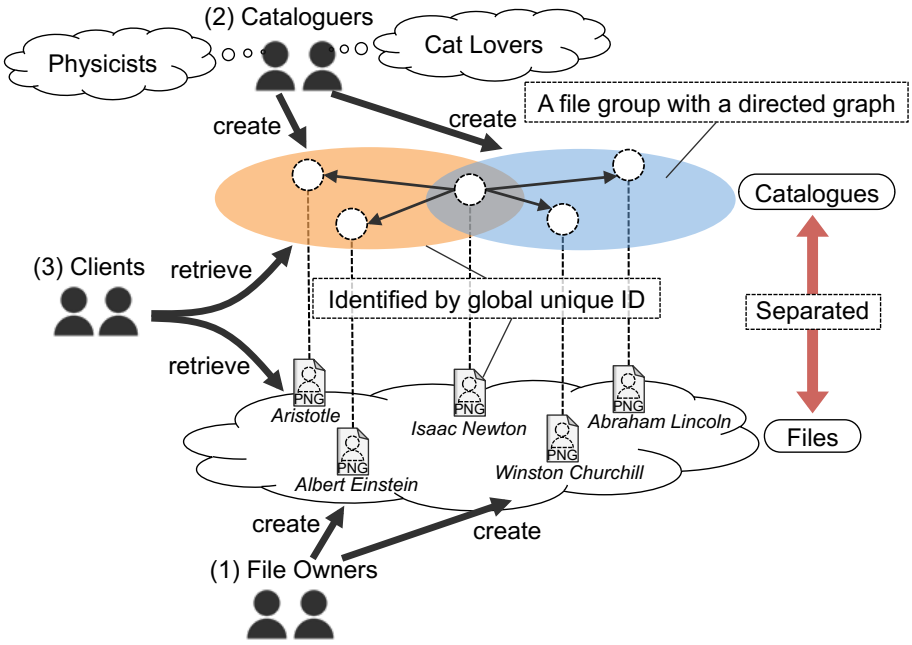


Figure 5.1: Overview of objects and users.

able to secure their catalogues as their assets. The cataloguers may entrust catalogue management to a service provider if it is technically difficult for the cataloguers to manage the equipment, the way some users choose mail hosting services such as Gmail, in which mail messages are managed by third-party servers.

Graph Manager

The Graph Manager controls mappings for reverse lookup and parts of catalogues. It is placed in the domain of the File Managers or the Catalogue Servers. The Graph Manager is responsible for managing two types of information: mappings for reverse lookup from a child object to parent catalogues, and edges connecting source and destination object vertex identifiers. A reverse lookup mapping consists of the child object identifier and the GCID of the parent catalogue, which enables clients to refer to all parent catalogues that contain the object of interest. When a cataloguer creates a new catalogue, an edge is pushed into two Graph Managers that manage the source and destination objects respectively. Thus, each Graph Manager has edges that originate from or terminate at objects in the domain of that Graph Manager. When a client retrieves an object, the client can retrieve all the edges that are connected to the object.

5.3 Design of Demitasse

5.3.1 Design overview

As noted above, the performance requirements of Demitasse are as follows: 1) Demitasse must play uncompressed UHD video stored in shared storage on the Internet; 2) Demitasse must be able to edit the order of component files and share them immediately without retrieving all the component files; 3) Demitasse must not require making duplicate copies of component files.

Demitasse uses Content Espresso to store video component files to satisfy requirements 1) and 3) and the Catalogue System to store and share the relationship among video component files to satisfy requirements 2) and 3). Figure 5.2 offers a visual overview of Demitasse. Initially, cameras shoot video content, divide it into component video files, and store those files with Content Espresso. Then, the cameras store the relationships among the files to the Catalogue System as catalogues. When a Demitasse Client, a client-side application of Demitasse, plays back stored video, it retrieves the catalogues from Catalogue System, obtains the GFIDs of the video component files, retrieves the video component files from Content Espresso, and plays them back one after another.

Demitasse can deal with any type of file as a video component file. Thus, Demitasse enables users to select the format of those files. For example, some producers may select a single uncompressed frame image file as their preferred video component file, while others some may select a single MPEG container file as their preferred video component file. In this design and implementation, Demitasse uses a single uncompressed frame image file as a single video component file to avoid any influence by encoding or decoding and compression or decompression in the Demitasse Client.

5.3.2 Demitasse Catalogue

Demitasse uses *Demitasse Catalogue*, a special structure of catalogues, to store and share the relations of video component files. Demitasse Catalogue should be designed to take the features of video component files into account. First, each video has metadata such as title, author, and frame rate that differs from video to video. In order to allow users to use flexible metadata description, the Demitasse Catalogue should have a GFID for the metadata file instead of managing the metadata directly as a property of the catalogue. Thus, users can write the required metadata into the metadata file in their own preferred fashion. Second, video content may have multiple angles, so the Demitasse Client should be able to select an angle when it is in playback mode. In order to deal with multi-angle video, Demitasse Catalogue thus needs a GFID of the file that contains the angle information for each video component file and the Catalogue ID of the catalogue that contains the GFIDs of the different angles. Finally, a user often wants to move through their video content both directions forward and backward rapidly. In order to start playback from any component files immediately in the Demitasse Client, the

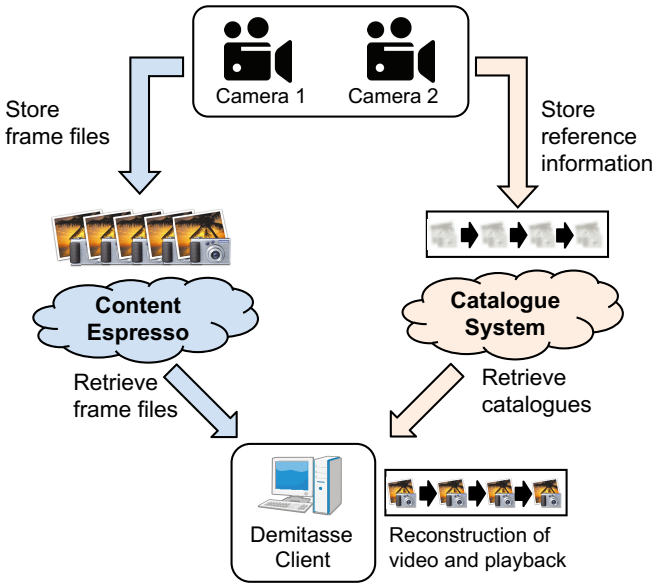


Figure 5.2: Overview of Demitasse. Demitasse uses Catalogue System to store the file relations between video component files and Content Espresso to store the video component files temselves.

Demitasse Catalogue should have a hierarchical structure that allows users to seek the GFID of a specific component file.

The Demitasse Catalogue is designed as shown in Figure 5.3. It is composed of five types of catalogue: *video metadata catalogue*, *video catalogue*, *segment catalogue*, *manual catalogue*, and *viewpoint catalogue*. Each circle in Figure 5.3 describes an object, either a GFID or a Global Catalogue ID (GCID). The first object of each catalogue is a GCID of that catalogue. A video metadata catalogue stores the GCID of a single video catalogue and a single GFID of the video metadata file. The video metadata file, written in XML, includes the title, author, frame rate, and total number of frames and is stored in a storage system such as a local file system or Content Espresso. A video catalogue stores the order of GCIDs of a segment catalogue. The last object in a video catalogue is the *endpoint object*, which indicates the end of the video. A segment catalogue includes the order of the GCIDs of manual catalogues; the number of manual catalogues inside a single segment catalogue can be determined by the creator of a particular Demitasse catalogue. In the current implementation, however, a single segment catalogue contains one second of video content. Thus, if the frame rate of the video is 30fps, each segment catalogue contains 30 manual catalogues. A manual catalogue contains a GCID of the viewpoint catalogue and a GFID of the manual file. The manual file, written in XML, has the metadata of the viewpoint catalogue, such as the number of viewpoints and the resolution of each viewpoint. The viewpoint catalogue stores one or more video component file GFIDs.

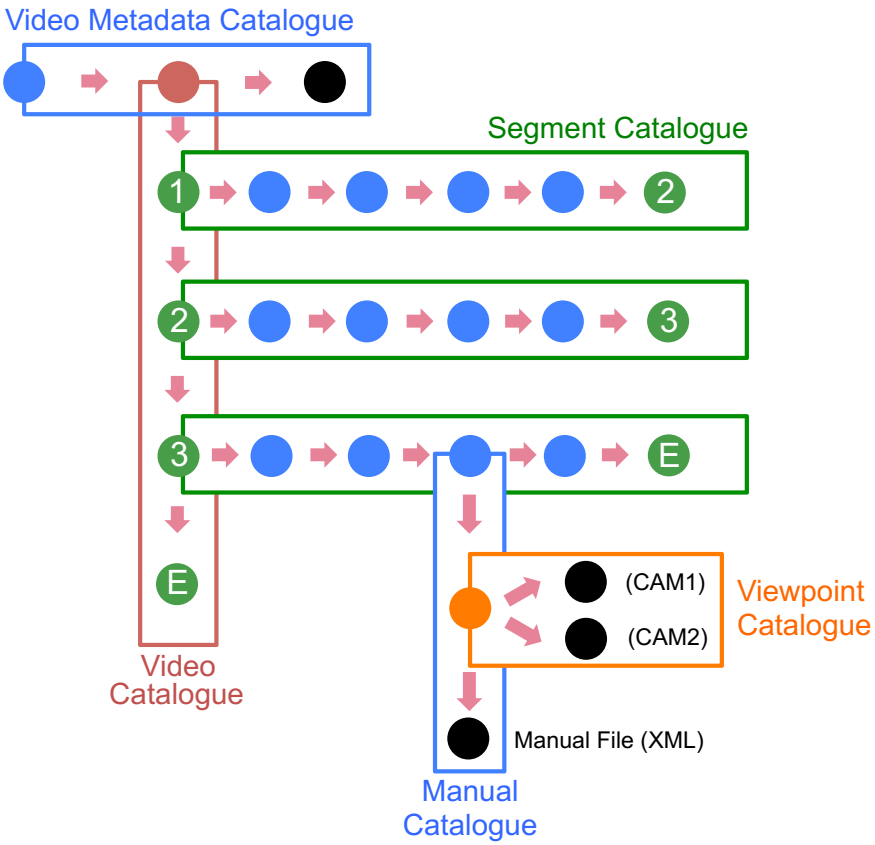


Figure 5.3: Video content description using Catalogue.

5.3.3 Demitasse Catalogue API

A Demitasse Catalogue has a complicated structure, which can make it difficult for users to seek and find a specific GFID by using the Catalogue Client API provided by Catalogue System. Therefore, Demitasse provides users with an API written in C++ to deal with Demitasse Catalogues more easily. The API has six main functions, as shown in Table 5.1.

The function `setMetaInfoCatalogue()` sets the GCID of the video content's video metadata catalogue. When this function is called, the function accesses the Catalogue Server and retrieves the requested video metadata catalogue by using Catalogue Client API. Then, the function acquires the GCID of the video catalogue and the GFID of the video metadata file from the video metadata catalogue. When `getMetaInfo()` is called, the function reads the video metadata file from the storage and parses it. Demitasse can access video metadata by using the `getMetaInfo()` function.

The `loadVideoCatalogue()` function is used to retrieve the video catalogue from the Catalogue Server by using the GCID in the video metadata catalogue. The video catalogue has multiple GCIDs for the segment catalogues. When the `start()`

Table 5.1: Demitasse Catalogue API functions.

Function	Argument
int setMetaInfoCatalogue	(**CATALOGID&** mICatId)
int getMetaInfo	(**char*** metaInfoString)
int start	()
bool next	(**FILEID&** fileId, **int** camId)
bool nextAll	(**FILEID*** fileId, **int** nCam)
int seek	(**int** n)

function is called, the function retrieves the first and second segment catalogues from the Catalogue Server. The function also parses the segment catalogues, acquires the GCIDs of the manual catalogues, and retrieves and parses the manual catalogues. Each manual catalogue contains the GCID of a viewpoint catalogue and the GFID of a manual file. When the manual catalogue is retrieved and parsed, the function acquires the viewpoint catalogue and the manual file.

The `next()` function is used to acquire the GFID of the next video component file. The second argument of the `next()` function is angle identifier. Thus, Demitasse can select the angle by changing the second argument. When the Demitasse requires the GFIDs of all angles, the `nextAll()` function can be used.

5.3.4 System modules

Demitasse is composed of Content Espresso, Catalogue System, and Demitasse Client. Content Espresso and the Catalogue System have been described in Chapter 2 and Section 5.2 respectively. The Demitasse Client, which is detailed below, is composed of four modules: *Frame Buffer*, *Catalogue Receiver*, *Frame Receiver*, and *Frame Viewer*. Figure 6.5 lays out the relationship among these modules.

Frame Buffer

Frame Buffer is a ring buffer designed for sharing GFID and frame image data among the Catalogue Receiver, Frame Receiver, and Frame Viewer. The size of the ring buffer can be determined by users.

Each Frame Buffer entry has one of the five following statuses: NO_FILEID, SET_FILEID, RECV_WAIT, RECV_FIN, or FEC_FAIL. NO_FILEID indicates that the Frame Buffer entry does not have the file ID of a video component file. SET_FILEID indicates that the Frame Buffer entry does have a valid video component file ID but Demitasse Client has not yet requested the file to Content Espresso. RECV_WAIT indicates that the Frame Buffer entry has a valid video component file ID and that Demitasse Client is retrieving the file from Content Espresso. RECV_FIN indicates the file from Content Espresso. RECV_FIN indicates that the Demitasse Client has retrieved the video component file from Content Espresso and that the retrieved data is valid. FEC_FAIL

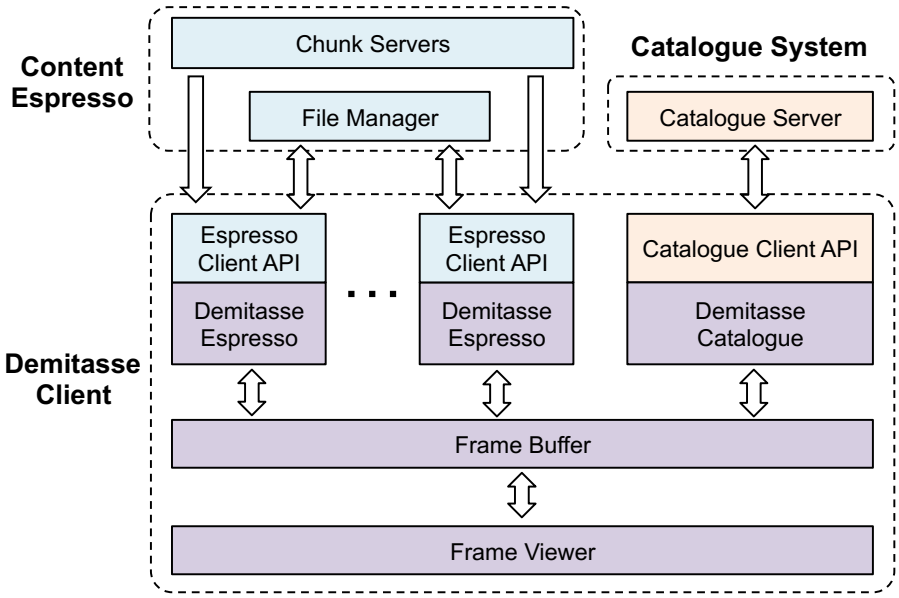


Figure 5.4: Demitasse has four modules: *Frame Buffer*, *Catalogue Receiver*, *Frame Receiver*, and *Frame Viewer*.

indicates that the Demitasse Client has retrieved the video component file from Content Espresso and that the retrieved data is invalid due to a failure in Content Espresso in recovering undelivered chunks.

Frame Buffer manages three pointers: SET_FILEID_POINTER, RECEIVE_FRAME_POINTER, and VIEW_FRAME_POINTER. SET_FILEID_POINTER indicates the frame buffer entry at which Catalogue Receiver should insert the next GFID. RECEIVE_FRAME_POINTER indicates the frame buffer entry at which Frame Receiver should obtain a GFID to send a file retrieval request to Content Espresso and store the retrieved file data. VIEW_FRAME_POINTER indicates the frame buffer entry at which Frame Viewer should obtain the video component file data for playback purposes.

Catalogue Receiver

The Catalogue Receiver accesses a Catalogue Server to retrieve each catalogue that composes the overall Demitasse Catalogue, acquires the GFID that the Demitasse Client should play back next, and inserts the GFID into the frame buffer entry that is indicated by the SET_FILEID_POINTER using the Demitasse Catalogue API. Figure 5.6 details the workflow of a Catalogue Receiver thread, which consists of an initial phase and a catalogue insertion phase. In the initial phase, the Catalogue Receiver sets the GCID of the MetaInfo catalogue by using the `setMetaInfoCatalogue()` function and retrieves the MetaInfo catalogue by using the `getMetaInfo()` function. The MetaInfo

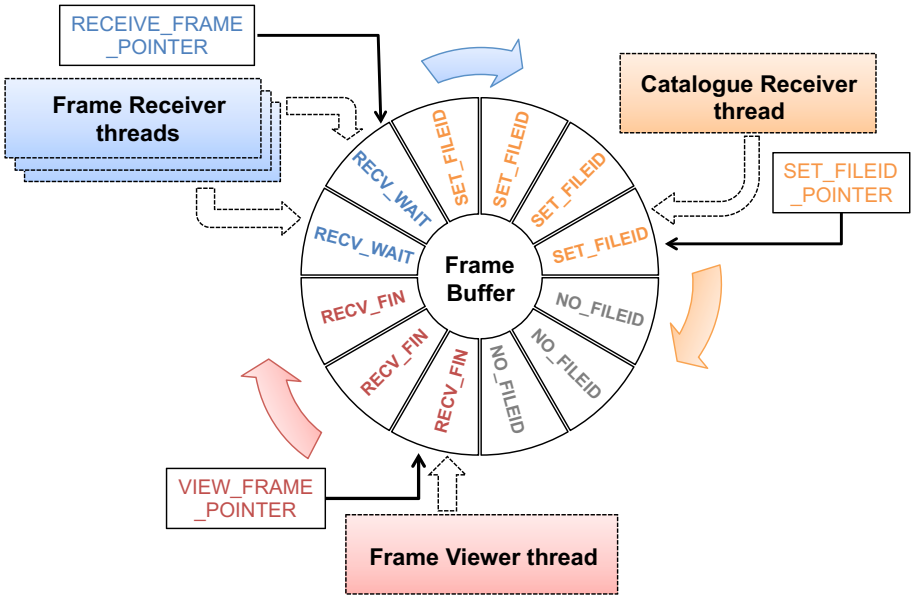


Figure 5.5: Frame Buffer and the status of each entry; Frame Buffer has set the SET_FILEID_POINTER, the RECEIVE_FRAME_POINTER, and the VIEW_FRAME_POINTER.

file that contains the metadata of the video is retrieved and parsed with the getMetaInfo() function, after which the metadata is shared with Frame Receiver, Frame Viewer, and Frame Buffer.

The catalogue insertion phase begins when the `start()` function is called and continues until the `next()` function in the Demitasse Catalogue API returns a value of -1. First, the Catalogue Receiver locks the SET_FILEID_MUTEX, acquires the pointer of the Frame Buffer entry that the SET_FILEID_POINTER points, increases the SET_FILEID_POINTER, and unlocks the SET_FILEID_MUTEX. Then, the Catalogue Receiver locks the BUFFER_MUTEX of the Frame Buffer entry and checks its status. If the status is not NO_FILEID, the Catalogue Receiver sleeps until the status becomes NO_FILEID. Otherwise, the Catalogue Receiver acquires the next GFID with the `next()` function, stores the GFID in the Frame Buffer entry, changes the status to SET_FILEID, unlocks the BUFFER_MUTEX, and notifies the Frame Receiver that the BUFFER_MUTEX has been unlocked. Finally, the Catalogue Receiver sleeps for a short time before continuing the catalogue insertion phase to adjust the GFID insertion rate. When the Frame Receiver works on multiple threads and the catalogue insertion rate becomes too rapid, simultaneous file retrieval requests to Content Espresso occur, which can cause chunk losses due to high network utilization.

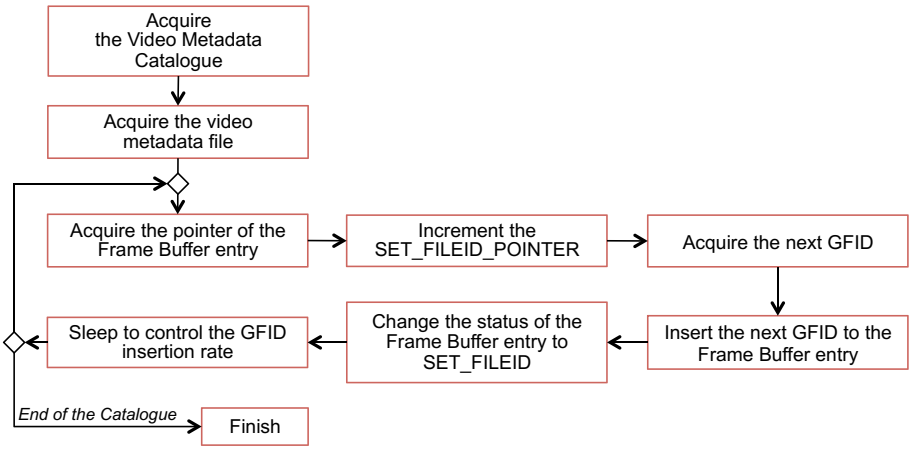


Figure 5.6: Flow chart of a Catalogue Receiver thread.

Frame Receiver

Frame Receiver acquires the GFID from the Frame Buffer entry indicated by RECEIVE_FRAME_POINTER, retrieves the frame image file from Content Espresso by using Content Espresso Client API and stores the retrieved frame image data in the Frame Buffer entry. Figure 5.7 details the workflow of a Frame Receiver thread, which consists of an initial phase and a frame retrieval phase. In the initial phase, the Frame Receiver establishes a connection with the Home File Manager, is authenticated by the Home File Manager, and sets up the parameters, such as chunk retrieval rate, for retrieving frame image files from Content Espresso.

The frame retrieval phase continues until the Catalogue Receiver finishes inserting the GFID. First, the Frame Receiver locks the RECVFRAME_MUTEX, acquires the pointer of the Frame Buffer entry that the FRAME_RECV_POINTER indicates, increases the FRAME_RECV_POINTER, and unlocks the RECVFRAME_MUTEX. Then, the Frame Receiver locks the BUFFER_MUTEX of the Frame Buffer entry and checks its status. If the status is not SET_FILEID, the Frame Receiver sleeps until the status becomes SET_FILEID. Otherwise, the Frame Receiver acquires the GFID from the Frame Buffer entry, allocates a buffer for the frame image file, and retrieves the frame image file from Content Espresso. If the complete frame image file is received, the Frame Receiver copies the received frame image data to the bitmap buffer in the Frame Buffer entry, and changes the status to RECV_FIN. Otherwise, the Frame Receiver changes the status to FEC_FAIL. Finally, the Frame Receiver unlocks the BUFFER_MUTEX and notifies the Frame Viewer that the BUFFER_MUTEX has been unlocked.

Frame Viewer

Frame Viewer acquires the frame image data from the Frame Buffer entry indicated by the VIEW_FRAME_POINTER and examines it, along with the frame rate written in the video metadata file. Figure 5.8 details the workflow of a Frame Viewer thread, which

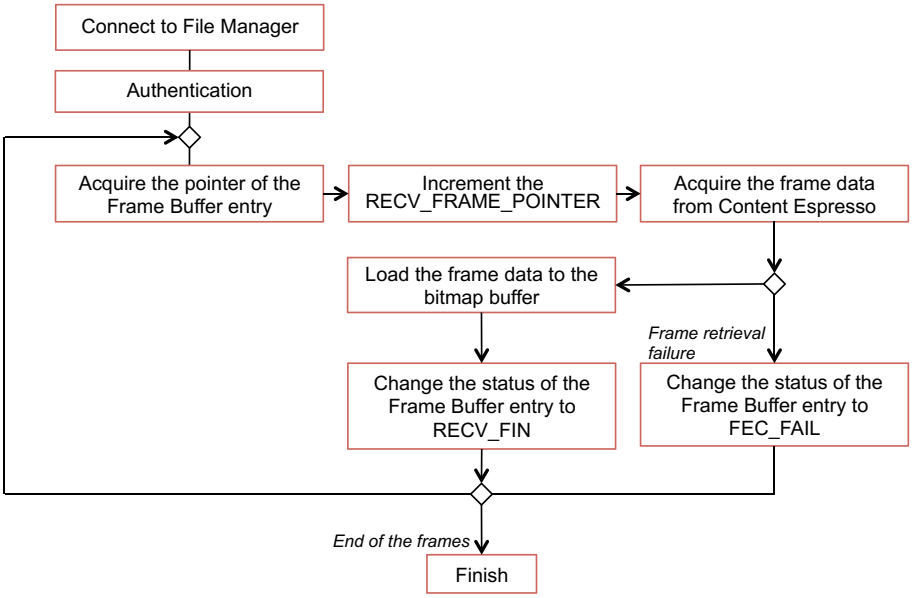


Figure 5.7: Flow chart of a Frame Receiver thread.

consists of an initial phase and a frame view phase. In the initial phase, the Frame Viewer acquires the frame rate from video metadata shared by the Catalogue Receiver, initializes OpenGL resources, and creates a window to view the frames. The frame view phase continues until the all frames have been viewed or the window is closed by the user.

First, the Frame Viewer acquires the pointer of the Frame Buffer entry indicated by the VIEW_FRAME_POINTER. Then, the Frame Viewer locks the BUFFER_MUTEX and checks the status of the Frame Buffer entry. If the status is RECV_FIN, the Frame Viewer displays the frame image data to the window. If the status is FEC_FAIL, the Frame Viewer skips to display the frame. After that, the Frame Viewer changes the status to NO_FILEID and unlocks the BUFFER_MUTEX. The Frame Viewer has to adjust the frame display timing to satisfy the specified frame rate; the method for adjusting the frame display timing is discussed in Subsection 5.3.6.

5.3.5 File retrieval interval

Demitasse Client has multiple Frame Receivers to retrieve frame files from Content Espresso. If simultaneous file retrieval requests to Content Espresso occur, network congestion can develop. Since Content Espresso uses UDP to retrieve chunks from Chunk Servers, network congestion can cause significant packet losses, which in turn leads to frame file retrieval failure in the Demitasse Client. Thus, Demitasse Client must have a mechanism to control the file retrieval interval.

In order to avoid excessive simultaneous requests to Content Espresso, the Catalogue Receiver controls the GFID insertion intervals. When there is no Frame Buffer entry

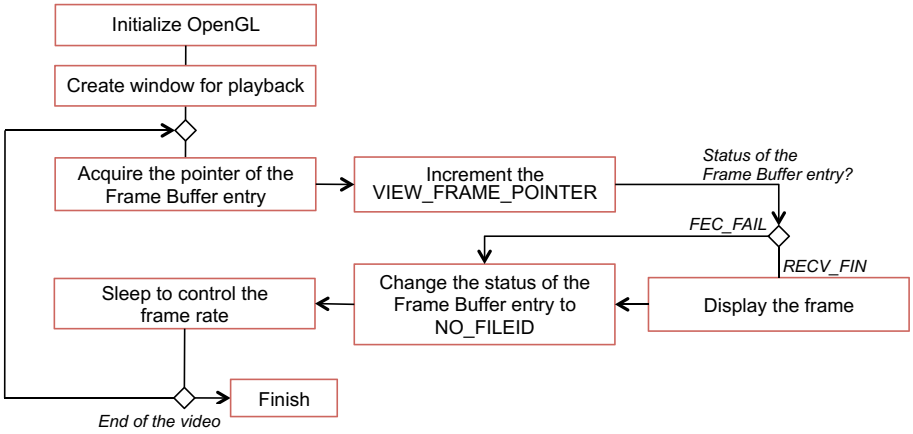


Figure 5.8: Flow chart of a Frame Viewer thread.

with a status of SET_FILEID and the Catalogue Receiver stops inserting GFIDs, the Frame Receiver sleeps until the Catalogue Receiver inserts a new GFID. Therefore, if the Catalogue Receiver inserts GFIDs at set intervals, the Frame Receiver will request frame files from Content Espresso at the same intervals. The request interval is determined by the size of the frame file S_{frame}, the chunk retrieval rate R_{recv}, and the frame rate R_{frame}. The request interval should be faster than the frame-viewing interval T_{view} and slower than the chunk retrieval time T_{recv}. T_{view} and T_{recv} are calculated as follows:

$$T_{view} = \frac{1}{R_{frame}} \tag{5.1}$$

$$T_{recv} = \frac{S_{frame}}{R_{recv}} \tag{5.2}$$

In the current implementation, the request interval is a little shorter than the frame viewing interval to avoid the starvation of retrieved frame files in the Frame Buffer. The Frame Buffer enables the Frame Viewer to continue to display video content even though frame retrieval may have stopped for a short time. After buffered frames have been consumed, the remainder of buffered frames becomes low and it is necessary to increase the buffered frames to deal with the next frame retrieval stopping. If the request interval is equal to the frame viewing interval, the number of buffered frames will no longer increase. Thus, the request interval $T_{request}$ is defined as follows where the buffer recovering rate is $R_{recover}$:

$$T_{request} = \frac{1}{R_{frame} + R_{recover}} \tag{5.3}$$

The buffer recovering rate $R_{recover}$ means an increase of $R_{recover}$ buffered files per second. For example, when $R_{recover}$ is five, five buffered frames increase every second until all Frame Buffer entries are occupied. $R_{recover}$ is determined by taking the available network bandwidth into account. In the current implementation, $R_{recover}$ is set to one. This buffer increasing mechanism is termed the buffer recovering mechanism.

5.3.6 Frame rate adjusting mechanism

The Frame Viewer plays back retrieved frame files with the frame rate defined in the MetaInfo file. The Demitasse Client uses OpenGL for displaying the frame files. Since OpenGL does not have a frame rate control mechanism, it is necessary to implement one. In order to adjust the frame rate, the Frame Viewer calculates the time when the next frame should be displayed and waits until that time arrives by using the `usleep()` function. First, the frame viewer obtains the base time t_{base} by using `gettimeofday()` when the frame viewer displays the first frame. Then, the Frame Viwer can calculate the time t_n when the n-th frame should be displayed. After that, the Frame Viewer calculates the sleep time T_{sleep} by using the current time. Both t_n and T_{sleep} can be calculated as follows by using the base time t_{base}, the frame rate R_{frame}, the current time $t_{current}$, and the frame number n:

$$t_n = t_{base} + \frac{n}{R_{frame}} T_{sleep} = t_n - t_{current} \tag{5.4}$$

5.3.7 Angle-switching mechanism

The Demitasse Client allows users to select specific angles in video content. The angle information is written in the Manual files by XML and are managed by Angle IDs, which are consecutive numbers starting at zero; the default Angle ID is thus zero. When the user changes the angle, the user inputs a specific Angle ID from an input device such as a keyboard. The Frame Viewer detects the input and sends the requested Angle ID to the Catalogue Receiver, which begins to select the file IDs that have the requested Angle ID and inserting them into Frame Buffer entries. After the Frame Receiver retrieves the new angle frame files, the Frame Viewer starts to display the requested angle frames. Since the Demitasse Client has multiple Frame Buffer entries, an angle-switching delay cannot be avoided.

5.4 Implementation of Demitasse

Demitasse Client is implemented with C++ and uses three external libraries and APIs: OpenGL for displaying frames, Catalogue Client API for accessing the Catalogue System to retrieve catalogues, and Content Espresso Client API for accessing Content Espresso to retrieve files.

OpenGL [18] is the computer industry's standard API for defining 2-D and 3-D graphic images. In order to use OpenGL with simple functions, GLFW [19] is used. GLFW is an open source, multiplatform library for OpenGL that provides a simple API for both creating windows, contexts, and surfaces and receiving input and events.

5.5 Evaluation of Demitasse

5.5.1 Evaluation overview

Demitasse is a network-oriented UHD video playback system using that uses Content Espresso and Catalogue System. The basic function of the Demitasse Client is retrieving video component files and viewing them at a specified frame rate. In its current implementation, the Demitasse Client supports a bitmap image file as a video component file. In this section, the file retrieval interval-adjusting mechanism, the frame rate control mechanism, and the Frame Buffer recovering mechanism of the Demitasse Client are all evaluated.

5.5.2 Experimental setup

In order to evaluate Demitasse, 72 Chunk Servers, a single File Manager, a single Storage Allocator, a single Catalogue Server, and a single Demitasse Client were set up. In addition, a tcpdump machine was also set up to capture packets. The tcpdump machine had 10Gbps NIC and was connected with an optical coupler to the same NETGEAR port switch that was connected to the Demitasse Client. An optical coupler can split transmission data from optical fibers. Thus, the tcpdump machine can capture all packets with a destination of the Demitasse Client. The reason for preparing the tcpdump machine in isolation from the Demitasse Client is to avoid having the tcpdump process occupy the CPU of the Demitasse Client. Figure 5.9 describes the Demitasse evaluation environment.

In order to evaluate Demitasse, two types of video content were used; uncompressed UHD and uncompressed Full HD. Each kind of content was split into uncompressed frame files and stored in Content Espresso. Table 5.2 shows the file type, file size, and encoded file size of each frame image file. The file type shows the format of the frame image file. In its current implementation, Demitasse can deal with bitmap files. File size shows the size of each frame file, and the encoded file size is the size of the original file size and the redundant data added by Content Espresso. These frame files are stored in Content Espresso with the parameters shown in Table 5.3. In addition, in order to play back the frame files in the Demitasse Client, five Demitasse Catalogues were created; UHD 15fps, UHD 30fps, Full HD 15fps, Full HD 30fps, and Full HD 60fps, as shown in Table 5.4. Since the available network bandwidth of Demitasse Client is 10Gbps, the maximum frame rate of UHD video is 30fps, and since the maximum frame rate of the Demitasse Client monitor is 60fps, the maximum frame rate of Full HD video is 60fps.

5.5.3 File retrieval interval

Demitasse determines the chunk retrieval rate when it retrieves the frame image files from Content Espresso. The chunk retrieval rate should be faster than the video throughput with FEC, as shown in Table 5.4. In this evaluation, the Demitasse Client plays back

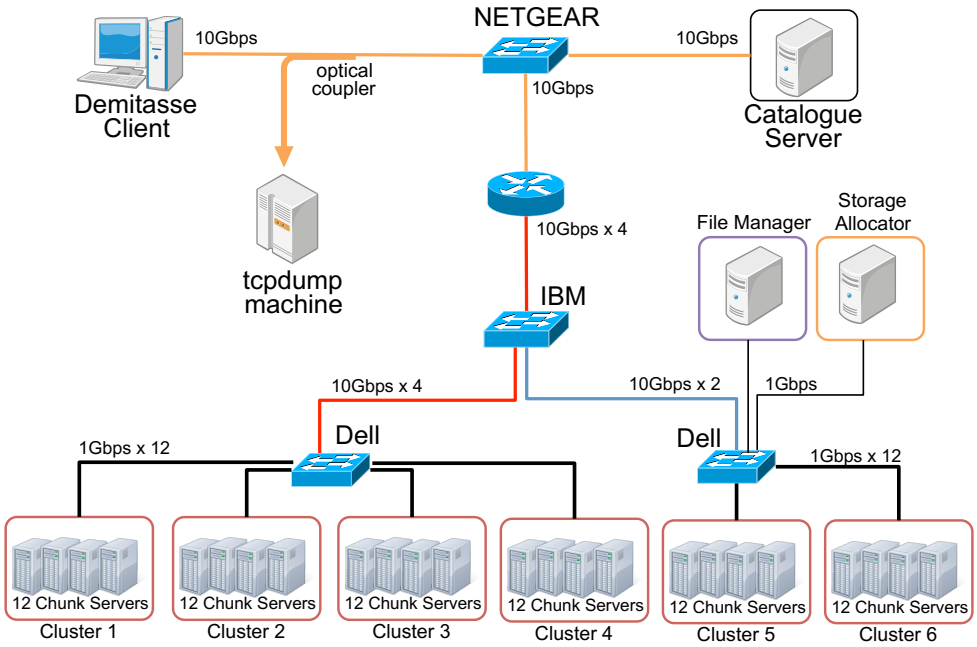


Figure 5.9: Experimental environment for evaluating Demitasse.

Table 5.2: Frame image specifications for the evaluation.

Image Size	**File Type**	**File Size**	**Enc File Size**
UHD (3840 x 2160)	24bit BMP	24,883,254 byte	29,971,254 byte
Full HD (1920 x 1080)	24bit BMP	6,220,854 byte	7,492,854 byte

Table 5.3: Content Espresso parameters when frame image files are stored.

Parameter Name	**Parameter**
Number of Clusters	6
Number of Chunk Servers	72
Chunk Size	1272
H_{Data}	1000
H_{Parity}	200

Full HD 15fps, Full HD 30fps, and Full HD 60fps videos with several chunk retrieval rates and captures the arrival chunks by using the tcpdump machine shown in Figure 5.9. After that, the captured data is analyzed and visualized to display the chunk arrival rate.

The Frame Receiver has to retrieve the video component files faster than video throughput with FEC. For example, the video throughput with FEC of Full HD 15fps

Table 5.4: Video specifications for the evaluation.

Image Size	Frame rate	Video throughput	Video throughput with FEC
UHD (3840 x 2160)	15fps	2.99Gbps	3.60Gbps
UHD (3840 x 2160)	30fps	5.97Gbps	7.19Gbps
Full HD (1920 x 1080)	15fps	747Mbps	899Mbps
Full HD (1920 x 1080)	30fps	1.49Gpbs	1.80Gbps
Full HD (1920 x 1080)	60fps	2.99Gbps	3.60Gbps

video is 899Mbps, so the Frame Receiver should set a chunk retrieval rate faster than 899Mbps to take the chunk retrieval overhead of Content Espresso, such as accessing File Manager, into due consideration.

In this evaluation, the chunk retrieval rates were set at 1Gbps, 2Gbps, 3Gbps, 4Gbps, and 5Gbps for Full HD 15fps video playback. Figure 5.10 presents the respective chunk arrival rates for one second; the file retrieval times become longer when the chunk retrieval rate is slower. This long chunk retrieval time might cause simultaneous file retrieval, which can lead to significant packet losses. Thus, the user should set a chunk retrieval rate that is much faster than the video throughput with FEC so as to achieve stable file retrieval.

The chunk retrieval rates were set at 3Gbps, 3.5Gbps, 4Gbps, 4.5Gbps, and 5Gbps in the evaluation of Full HD 30fps, and at 4Gbps, 4.5Gbps, 5Gbps, 5.5Gbps, and 6Gbps in the evaluation of Full HD 60fps, taking the video throughput with FEC into consideration. Figures 5.11 and Figure 5.12 present the respective chunk arrival rates for one second of those videos. The results show a trend that is similar to what was found with Full HD 15fps video. The overall evaluation results confirm that the file retrieval interval mechanism works correctly.

5.5.4 Frame rate control

Demitasse Client's frame rate control mechanism, described in Subsection 5.3.6, was evaluated by measuring the average frame rate every second in Full HD 15fps, Full HD 30fps, Full HD 60fps, UHD 15fps, and UHD 30fps video playback with 2Gbps, 4Gbps, 6Gbps, and 8Gbps retrieval rates. Figures 5.13 and 5.14 present the results for 100 seconds of Full HD and UHD video respectively. The average frame rate of the first period is faster than the expected frame rate in all measurements, because frame rate controlling does not work for the first several frames in the current Demitasse implementation. After this initial period, the frame rate remains stable at all retrieval rates in Full HD video playback. Therefore, these evaluation results confirm that the frame rate control mechanism works well for Full HD video playback. In UHD 15fps video playback, the frame rate is stable after the first period at any retrieval rate. However, in UHD 30fps

video playback, the frame rate is comparatively unstable at any retrieval rate, which results from much of the CPU's resources being utilized for retrieving chunks. As a result, Frame Viewer as implemented by OpenGL becomes a little unstable in displaying each frame image. Nevertheless, the frame rate remains between 29 and 31. Therefore, the results of this evaluation confirm that the frame rate control mechanism works well, overall, in UHD video playback.

5.5.5 Frame Buffer status

Demitasse has a Frame Buffer for sharing GFIDs between the catalogue thread and frame recv threads, store retrieved frame files from Content Espresso, and send frame files to the frame play thread. A Frame Buffer has 30 entries, each of which has one of the following statuses: NO_FILEID, SET_FILEID, RECV_WAIT, RECV_FIN, FEC_FAIL, or VIEW_WAIT.

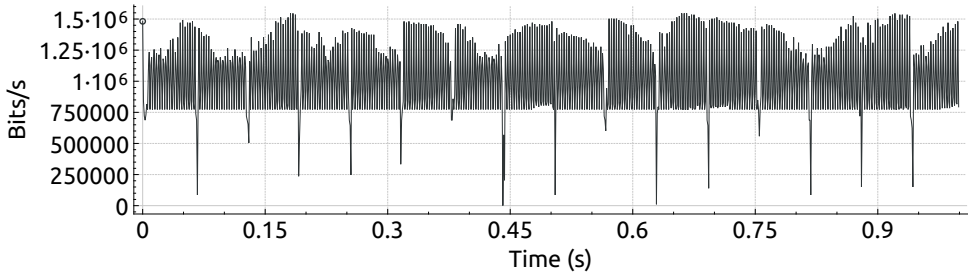
In this evaluation, Demitasse played back Full HD 15fps, Full HD 30fps, Full HD 60fps, UHD 15fps, and UHD 30fps video at 2Gbps, 4Gbps, 6Gbps, and 8Gbps retrieval rates. The number of different statuses of Frame Buffer entries and the total number of lost frames were measured every second for the first 100 seconds, and the results of those measurements were visualized. Figures 5.15 to 5.17 present the measurements of Full HD video playback and Figures 5.18 and 5.19 present the measurements of UHD video playback. The x-axis is the number of frames, the right y-axis is the status of each Frame Buffer entry, and the left y-axis is the number of lost frames. In Full HD video playback, no frame files were lost during the measurement.

In the first measurement in Figure 5.15 (a), for example, 27 Frame Buffer entries had NO_FILEID status, one Frame Buffer entry had RECV_WAIT status, and two Frame Buffer entries had RECV_FIN status. In its current implementation, the Demitasse Client starts playing back the video as soon as the first frame file arrives; it does not wait until the Frame Buffer is filled. Thus, 27 of 30 Frame Buffer entries are empty and only two frames are buffered at the first measurement. The number of buffered frames increases every measurement because the Demitasse Client has the Frame Buffer recovering mechanism discussed in Subsection 5.3.5. Since the buffer recovering rate is set at one in the current parameters, there is an increase of one buffered frame with each measurement. This phase is called the buffering phase. After the buffering phase, 29 of 30 Frame Buffer entries take RECV_FIN status and the number of RECV_FIN entries do not change. This phase is called the stable phase.

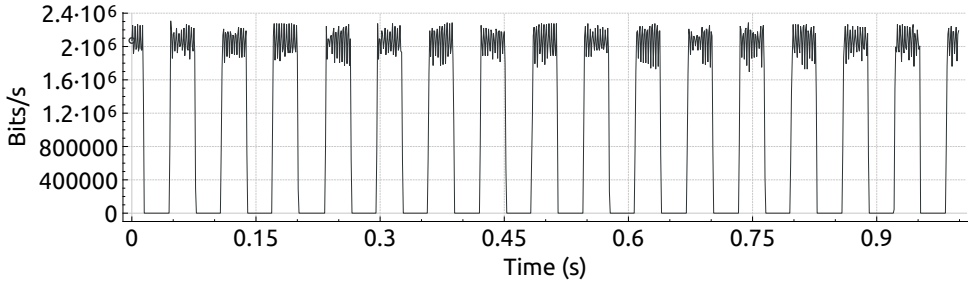
Frame Buffer entries with SET_FILEID appeared in only one measurement shown in Figure 5.17 because the Demitasse Client measures the status of the Frame Buffer entries every second, but the Frame Receiver changes the status of the Frame Buffer entry to RECV_WAIT as soon as the Frame Receiver finds the SET_FILEID status entry and the Demitasse Client has a sufficient number of Frame Receivers.

The number of RECV_WAIT entries increases when the frame rate increases, the file retrieval rate decreases, and the frame size increases. For example, while the number of RECV_WAIT entries is zero in the stable phase of Full HD 15fps playback with a

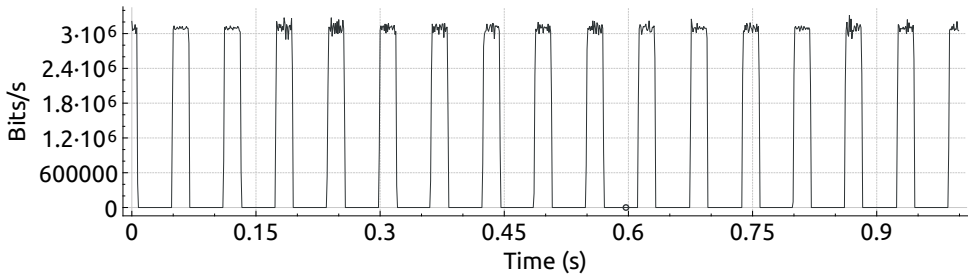
2Gbps retrieval rate, it is two in the stable phase of Full HD 60fps playback at a 2Gbps retrieval rate, as shown in Figure 5.15 (a) and Figure 5.17 (a). While there is only one RECV_WAIT entry is one in the stable phase of UHD 30fps playback at an 8Gbps retrieval rate, there are four in the stable phase of UHD 30ps playback at a 2Gbps retrieval rate, as shown in Figure 5.19 (d) and Figure 5.19 (a). While the number of RECV_WAIT entries is zero in the stable phase of Full HD 15fps playback at a 2Gbps retrieval rate, it is two in the stable phase of UHD 15fps playback at a 2Gbps retrieval rate, as shown in Figure 5.15 (a) and Figure 5.18 (a). It is necessary to retrieve frame files with higher throughput when the frame rate increases, when the file retrieval rate decreases, and when the frame size increases. Thus, the Demitasse Client has to retrieve the frame files using multiple Frame Receiver threads. Since the number of RECV_WAIT entries is the number of Frame Receiver threads working simultaneously, the number of RECV_WAIT entries increase when the frame rate increases, when the file retrieval rate decreases, and when the frame size increases. High throughput file retrieval causes network congestion that can lead to frame file retrieval failures. In this experiment, frame loss only occurred in UHD 30fps video playback at retrieval rates above 4Gbps. In order to achieve stable file retrieval and playback, it is necessary to determine the appropriate file retrieval rate.



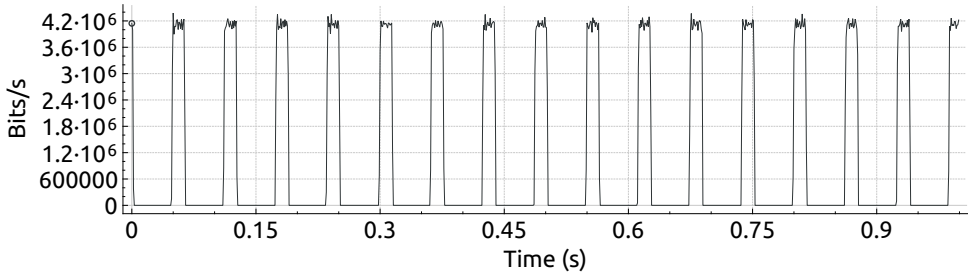
(a) Retrieval rate is 1Gbps.



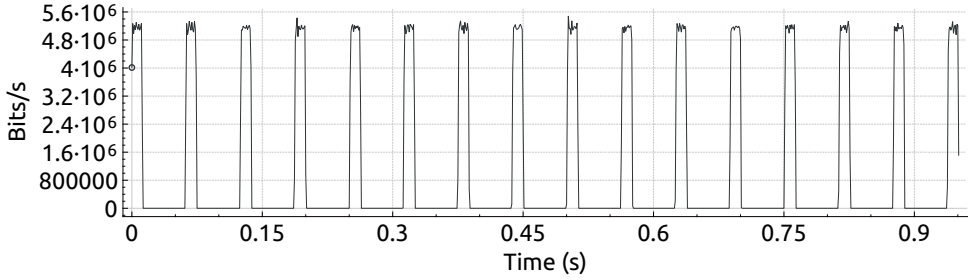
(b) Retrieval rate is 2Gbps.



(c) Retrieval rate is 3Gbps.

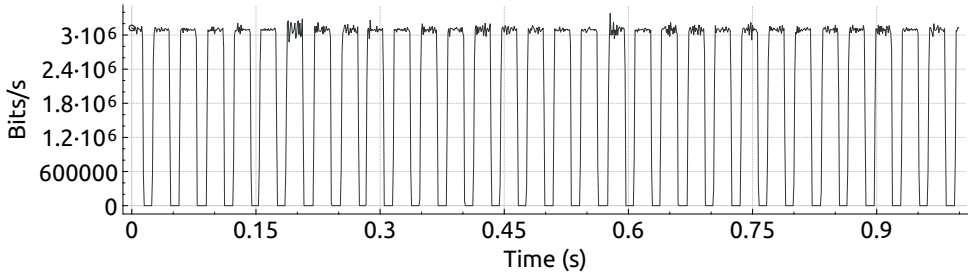


(d) Retrieval rate is 4Gbps.

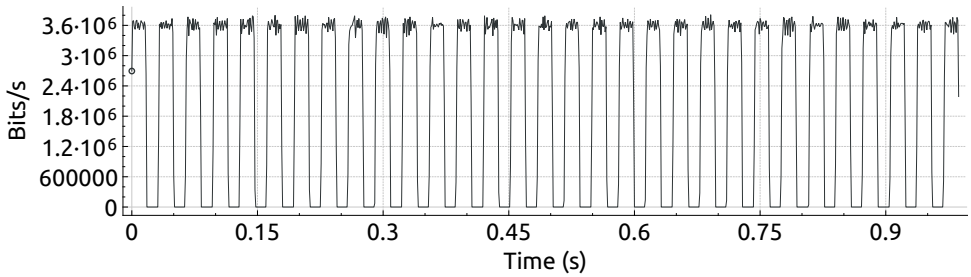


(e) Retrieval rate is 5Gbps.

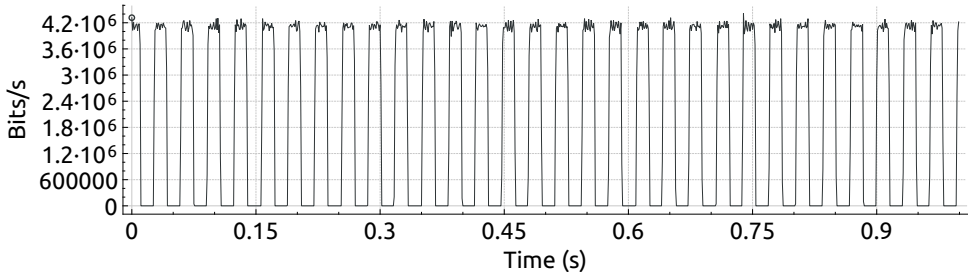
Figure 5.10: Chunk arrival rate in Full HD at 15fps.



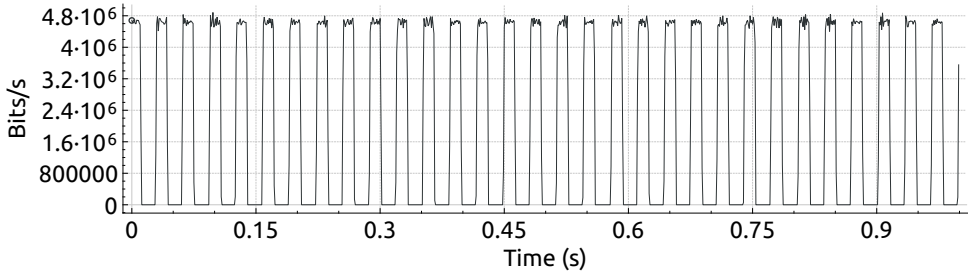
(a) Retrieval rate is 3Gbps.



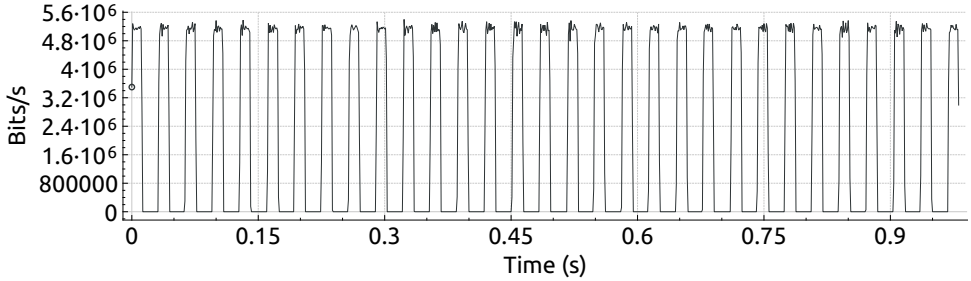
(b) Retrieval rate is 3.5Gbps.



(c) Retrieval rate is 4Gbps.

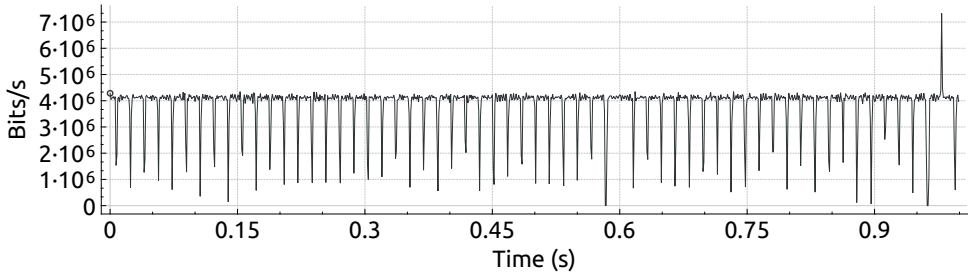


(d) Retrieval rate is 4.5Gbps.

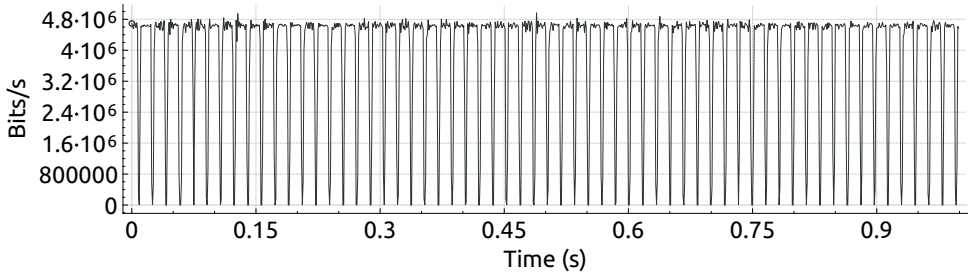


(e) Retrieval rate is 5Gbps.

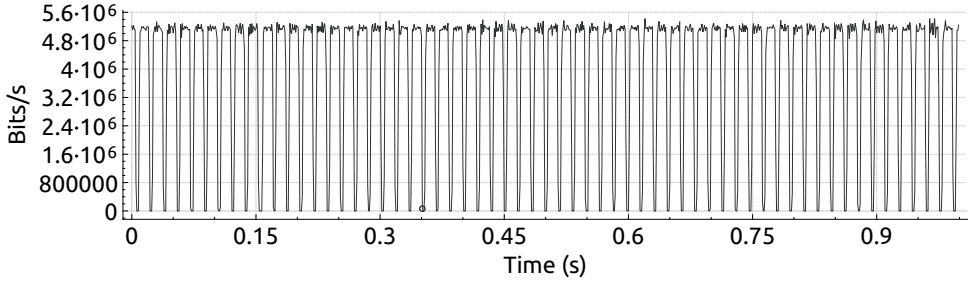
Figure 5.11: Chunk arrival rate in Full HD at 30fps.



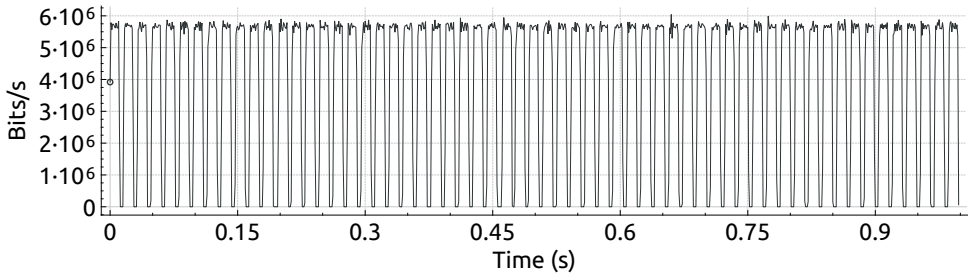
(a) Retrieval rate is 4Gbps.



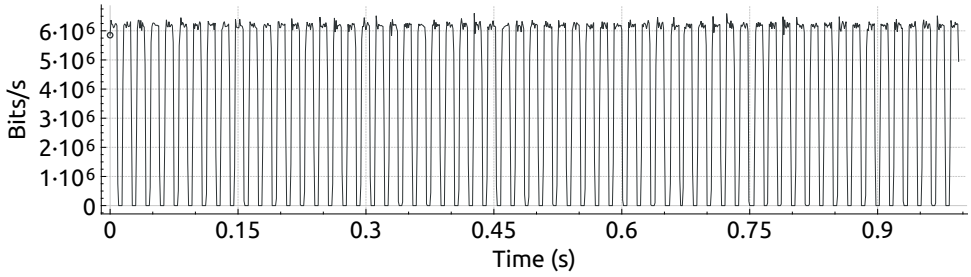
(b) Retrieval rate is 4.5Gbps.



(c) Retrieval rate is 5Gbps.

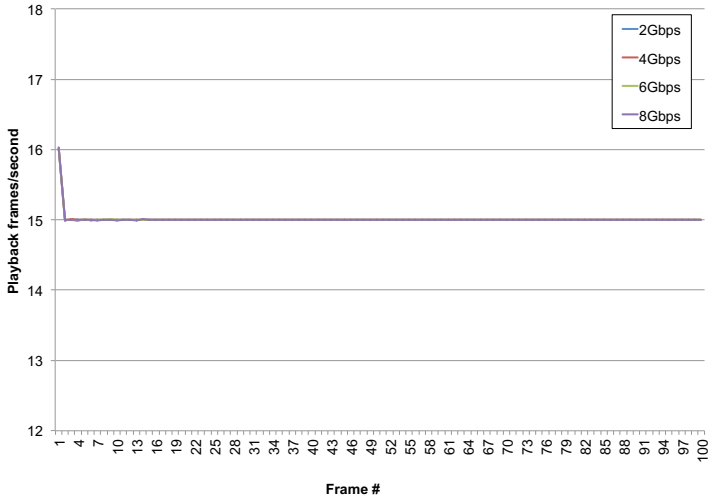


(d) Retrieval rate is 5.5Gbps.

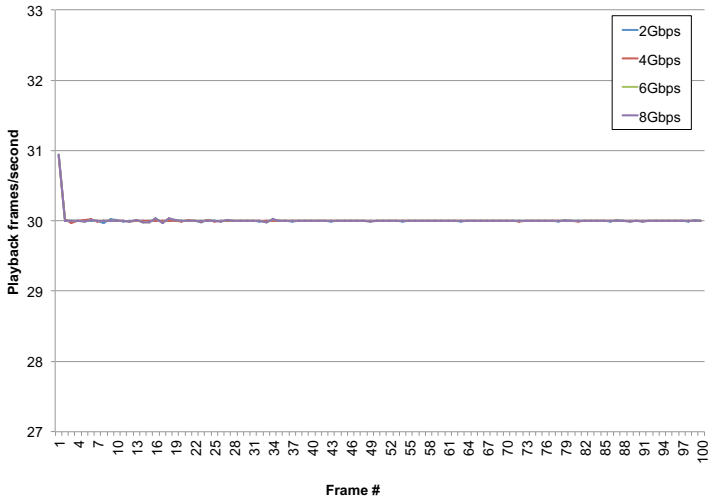


(e) Retrieval rate is 6Gbps.

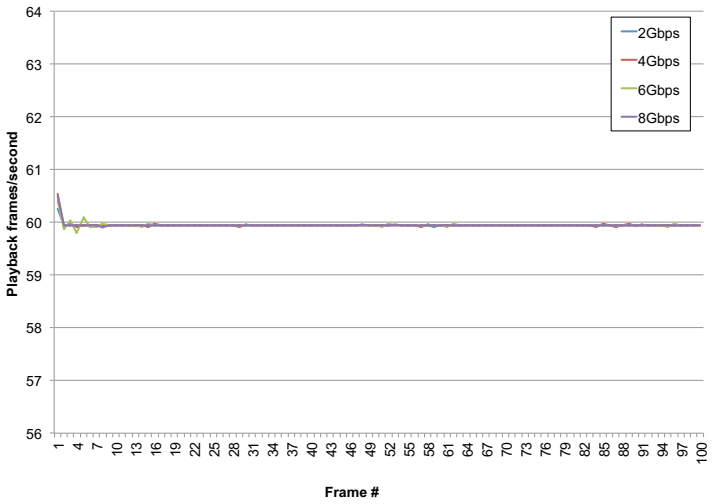
Figure 5.12: Chunk arrival rate in Full HD at 60fps.



(a) Full HD 15fps.

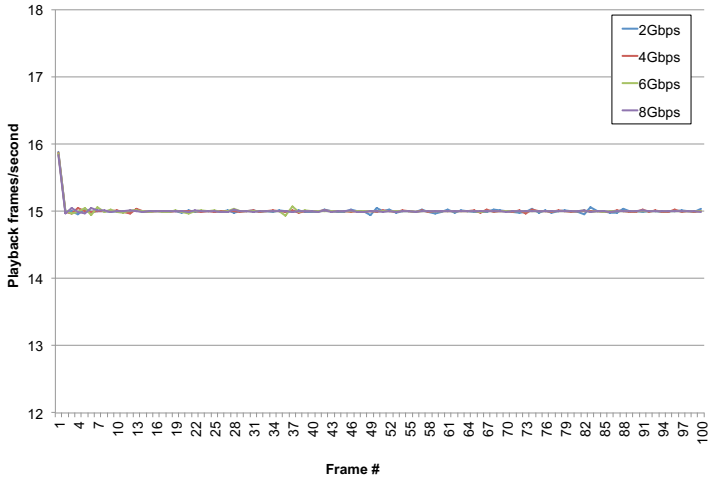


(b) Full HD 30fps.

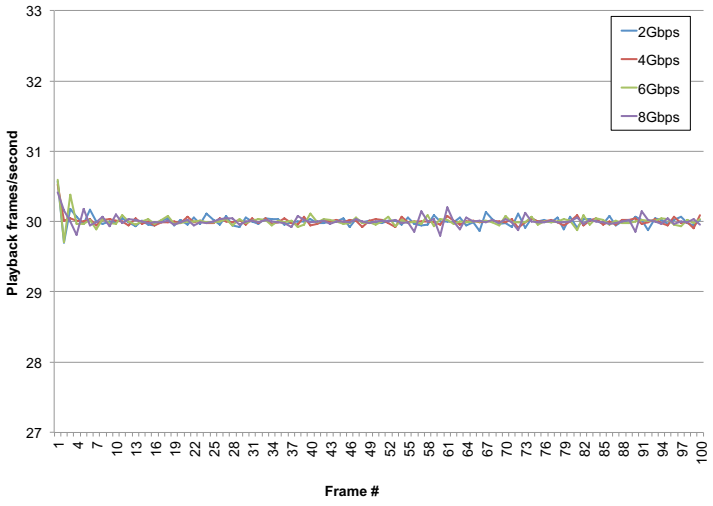


(c) Full HD 60fps.

Figure 5.13: Frame rate stability in Full HD playback.

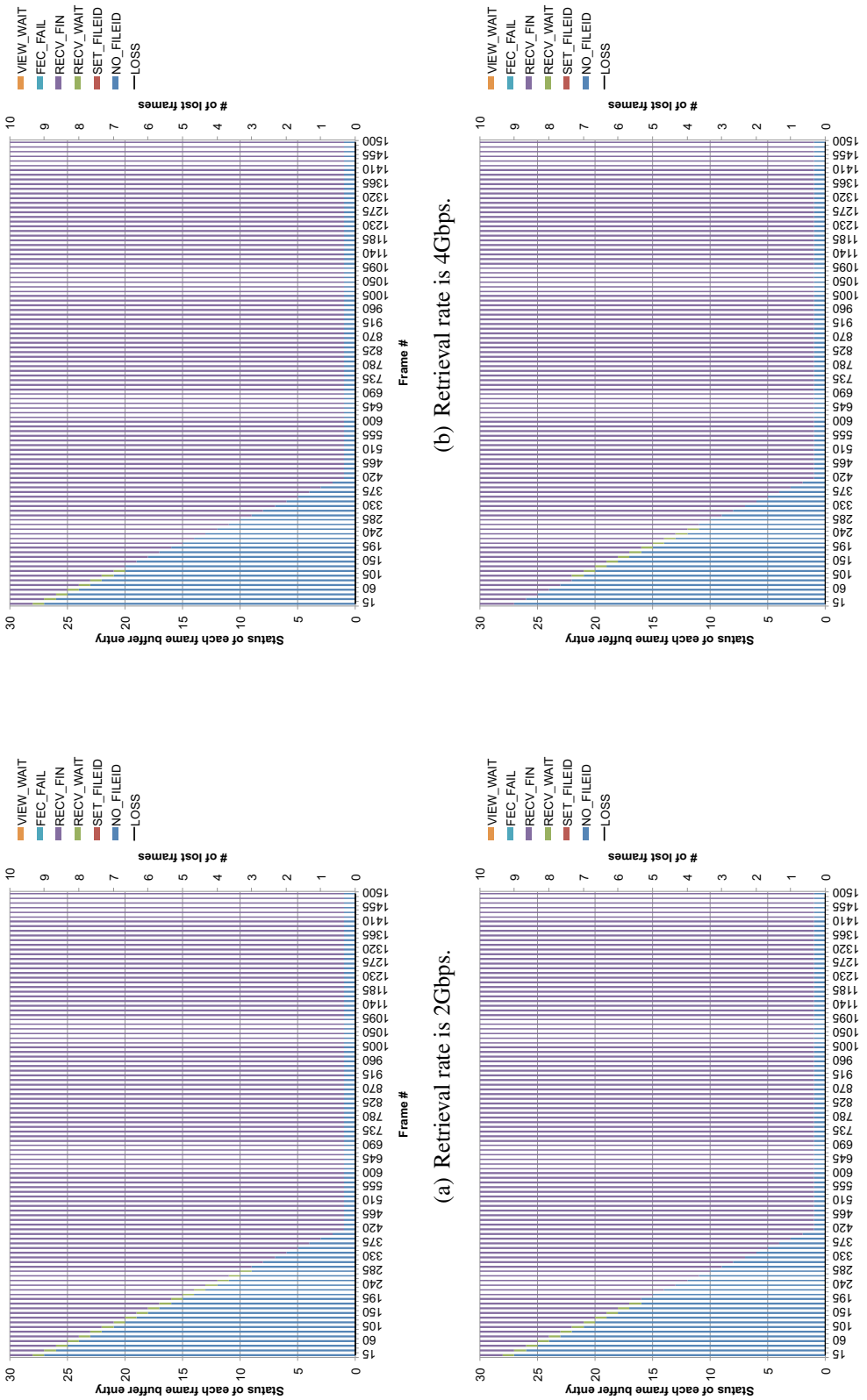


(a) UHD 15fps.



(b) UHD 30fps.

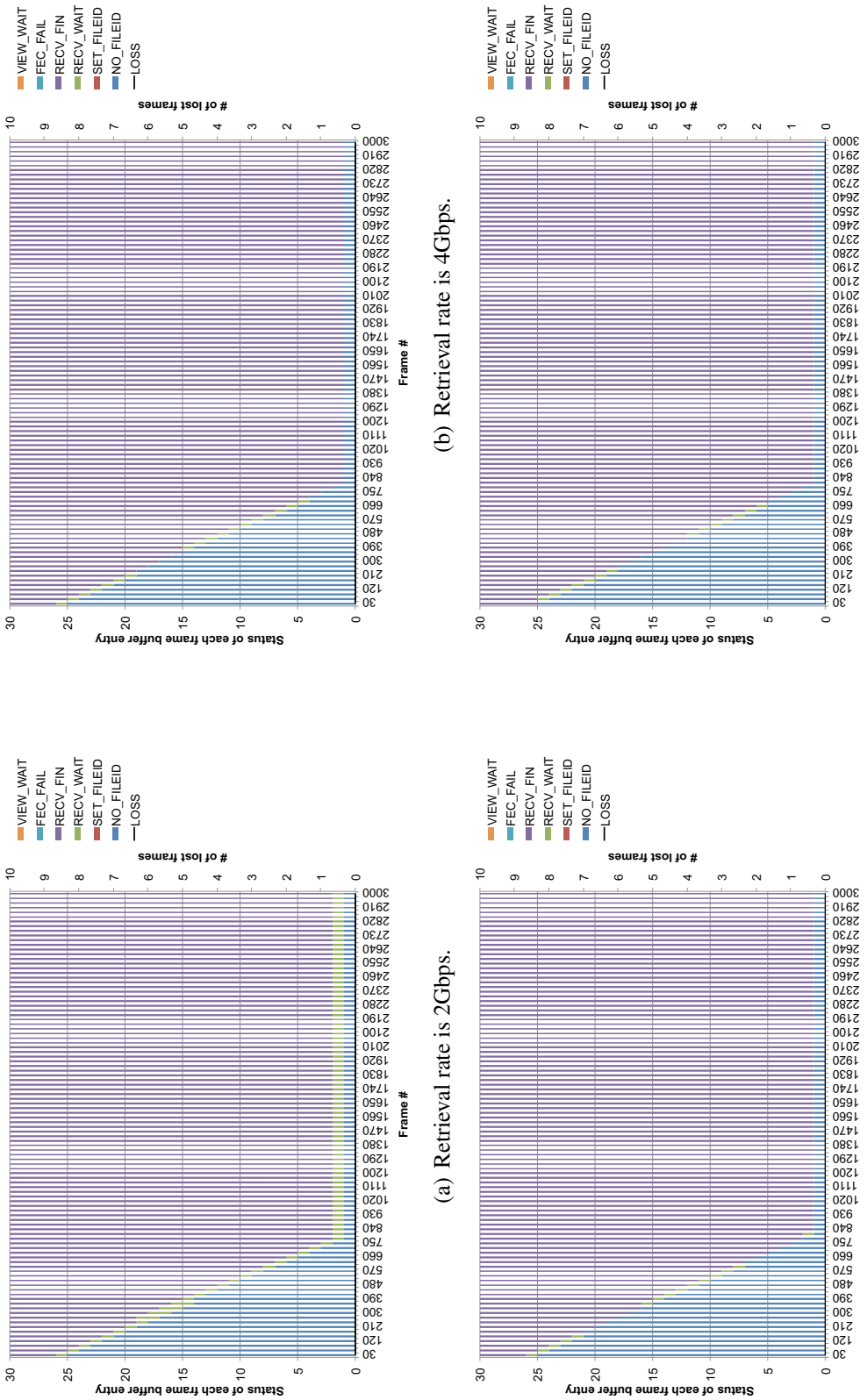
Figure 5.14: Frame rate stability in UHD playback.



(a) Retrieval rate is 2Gbps. (b) Retrieval rate is 4Gbps.

(c) Retrieval rate is 6Gbps. (d) Retrieval rate is 8Gbps.

Figure 5.15: Frame buffer status in 15fps Full HD video playback with 2 - 8 Gbps retrieval.



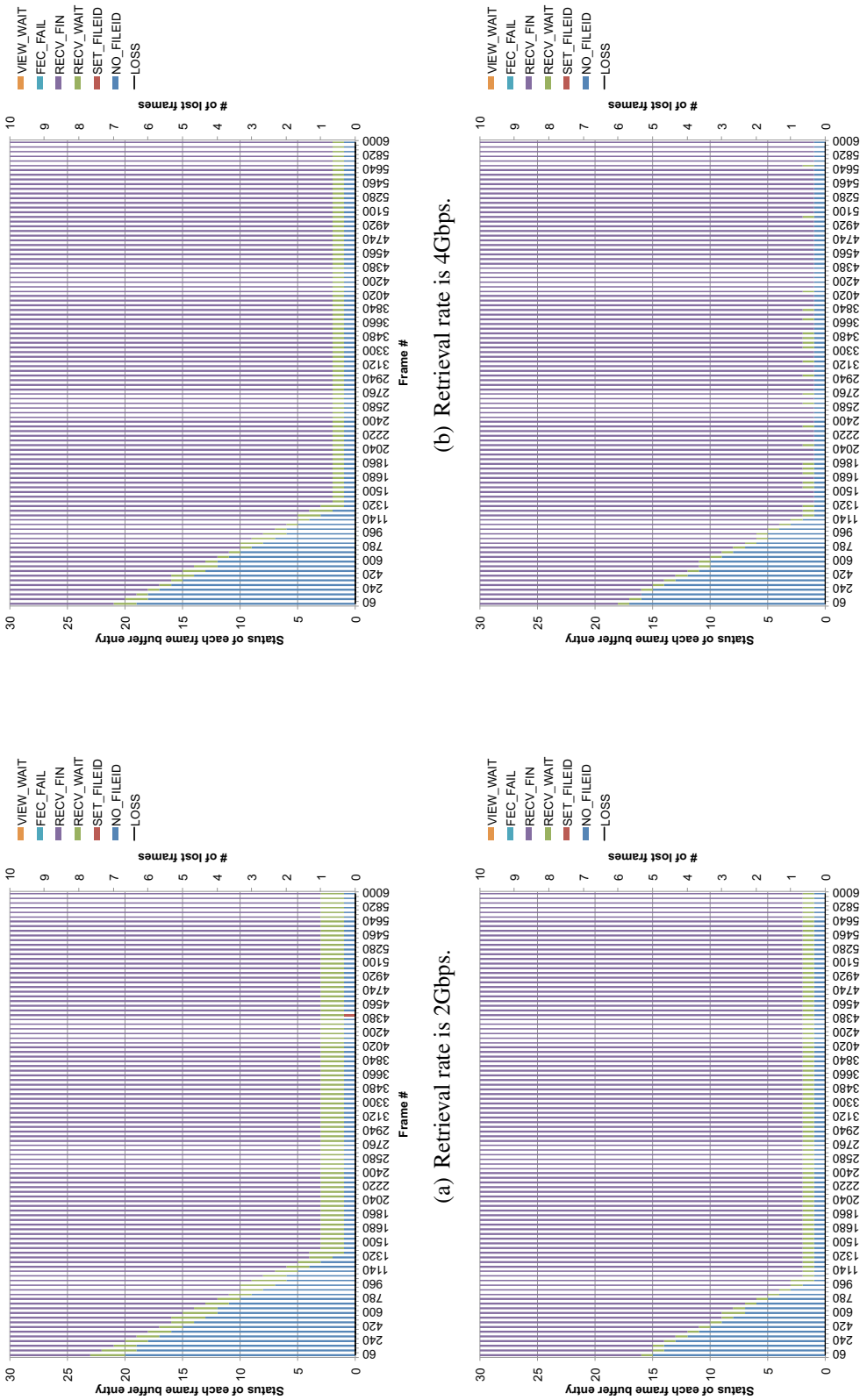
(a) Retrieval rate is 2Gbps.

(b) Retrieval rate is 4Gbps.

(c) Retrieval rate is 6Gbps.

(d) Retrieval rate is 8Gbps.

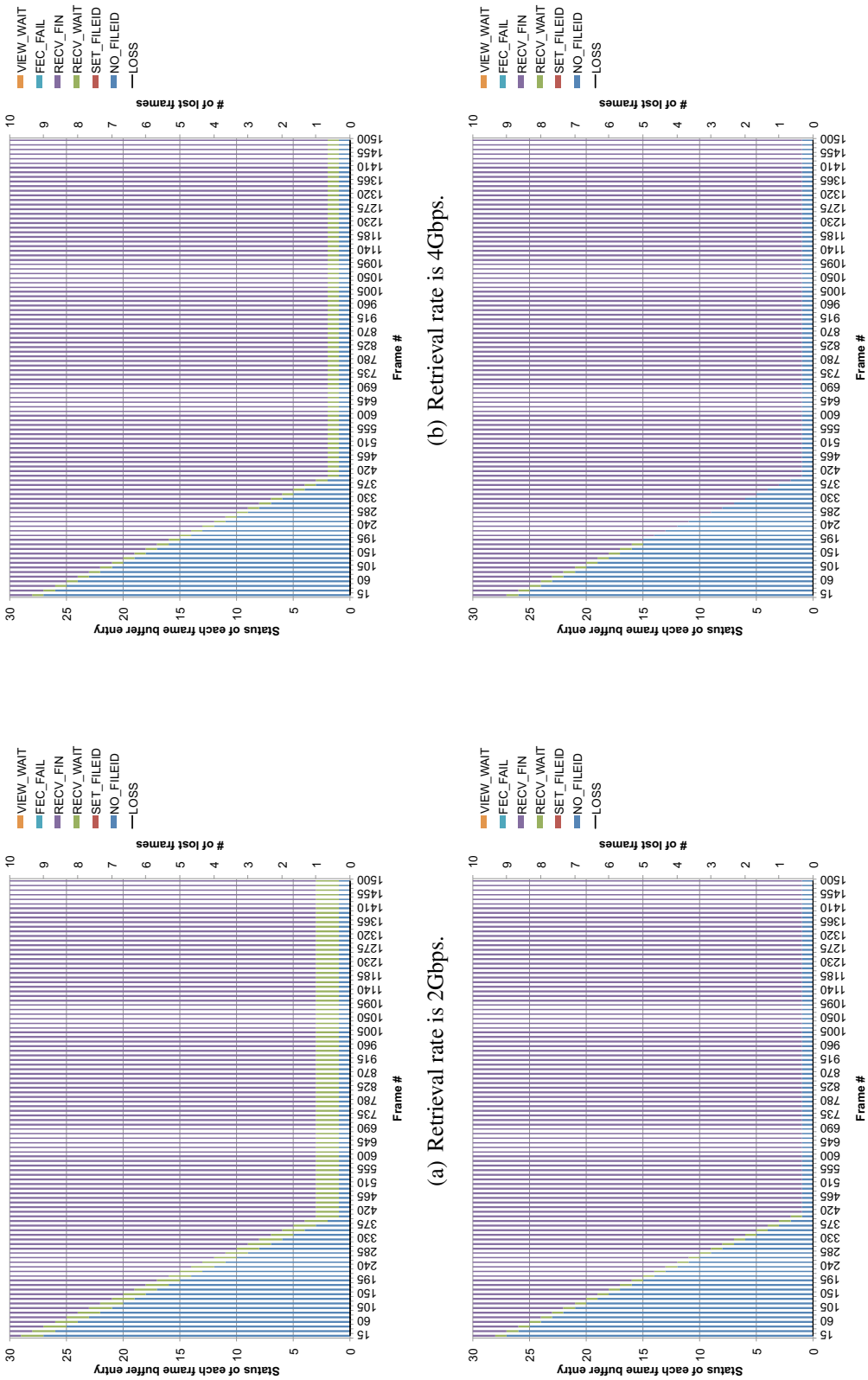
Figure 5.16: Frame buffer status in 30fps Full HD video playback with 2 - 8 Gbps retrieval.



(a) Retrieval rate is 2Gbps. (b) Retrieval rate is 4Gbps.

(c) Retrieval rate is 6Gbps. (d) Retrieval rate is 8Gbps.

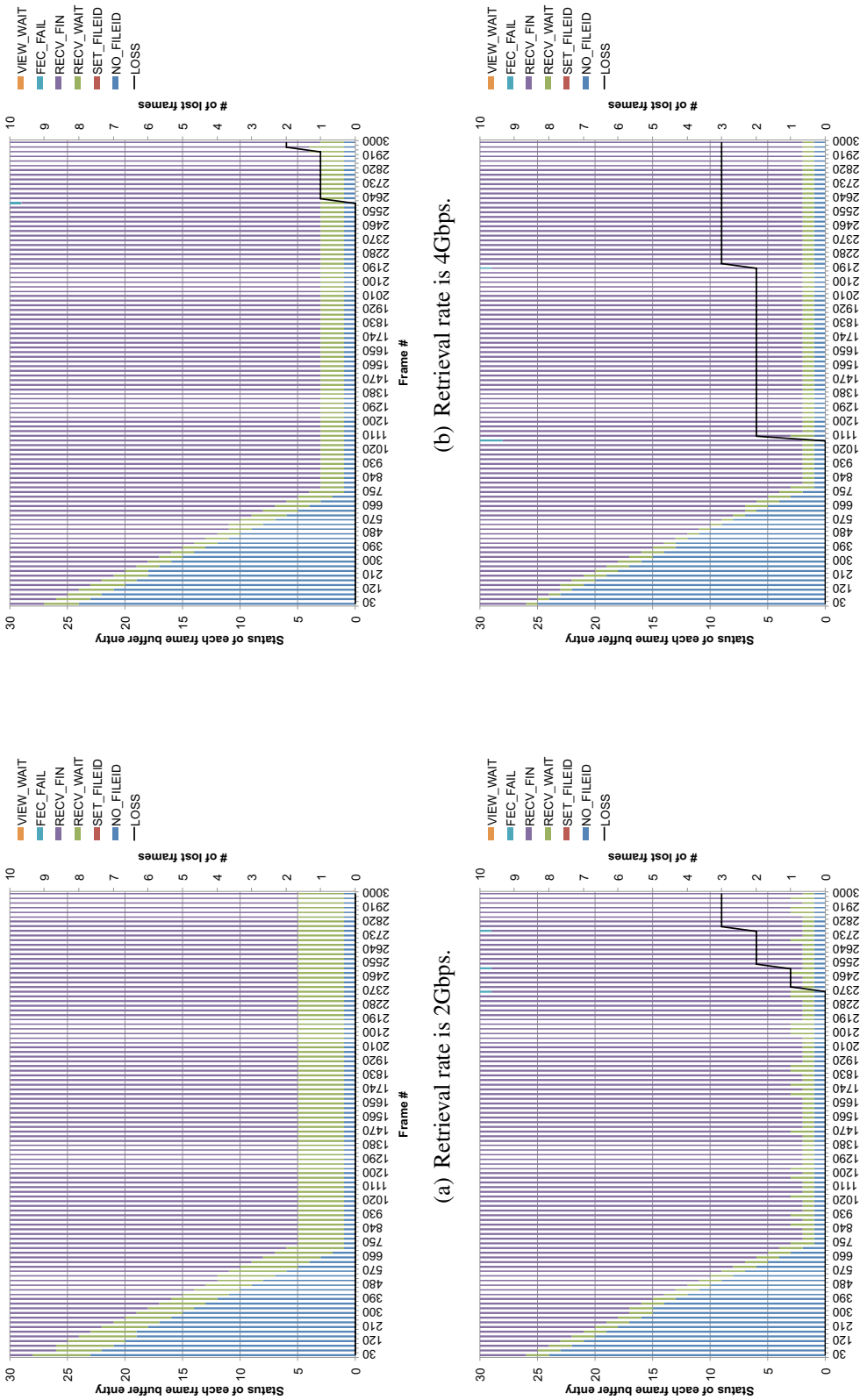
Figure 5.17: Frame buffer status in 60fps Full HD video playback with 2 - 8 Gbps retrieval.



(a) Retrieval rate is 2Gbps. (b) Retrieval rate is 4Gbps.

(c) Retrieval rate is 6Gbps. (d) Retrieval rate is 8Gbps.

Figure 5.18: Frame buffer status in 15fps UHD video playback with 2 - 8 Gbps retrieval.



(a) Retrieval rate is 2Gbps. (b) Retrieval rate is 4Gbps.

(c) Retrieval rate is 6Gbps. (d) Retrieval rate is 8Gbps.

Figure 5.19: Frame buffer status in 30fps UHD video playback with 2 - 8 Gbps retrieval.

5.6 Summary

This chapter introduced Demitasse, a file-based video content playback system using Content Espresso, and Catalogue System, as an example of Content Espresso applications. Demitasse stores video component files to Content Espresso and the relations among those files as a Demitasse Catalogue to Catalogue System. The Demitasse Client, which is a client-side application of Demitasse, retrieves the Demitasse Catalogue from the Catalogue System, parses it, obtains the appropriate GFIDs, retrieves the corresponding video component files from Content Espresso, and plays them back at the requested frame rate. The Demitasse Client is composed of four modules: Frame Buffer, Catalogue Receiver, Frame Receiver, and Frame Viewer. The Demitasse Client is implemented with Content Espresso API, Catalogue API, and OpenGL. In order to achieve stable file retrieval and playback, a frame rate adjusting mechanism and a file retrieval interval controlling mechanism are implemented. Although Demitasse is designed to deal with any type of vide component files, the Demitasse Client supports a bitmap image file as a video component file in its current implementation.

In order to evaluate the frame rate adjusting mechanism and the file retrieval interval controlling mechanism, 79 physical machines including 72 Chunk Servers were prepared to store video component files and the Demitasse Catalogue, and for playback of the stored files. The results of the evaluation confirmed that both the frame rate adjusting mechanism and the file retrieval interval controlling mechanism work correctly and that Demitasse Client can play back, with acceptable stability, uncompressed Full HD video files at up to 60fps and uncompressed UHD video files at up to 30fps. These results also mean that Content Espresso has enough storage IO performance to distribute uncompressed UHD video content.

Chapter 6

Improvement of File Sharing Performance of Web-Based Collaboration Systems

6.1 Background

With the explosive growth in demand for remote collaboration over the Internet, many web-based collaboration systems have been developed to support such work; examples include Google Apps [20] and SAGE2 [2]. SAGE2 is a web-based collaboration system that shares information on large high-resolution displays with other sites and enables people in multiple locations to work together and view the same displays. Although SAGE2 is useful for remote collaboration, it takes a long time to share large files like high-resolution images between SAGE2 systems that are located far apart from each other, because SAGE2 utilizes HTTP, a TCP-based protocol, to transmit files. Content Espresso, a distributed storage system for global large file sharing, is designed and implemented to provide a global high throughput large file sharing mechanism. Content Espresso enables low cost data storage and high throughput file transmission regardless of client location by using FEC and UDP, as described in Chapter 2.

This chapter aims to provide a high throughput file sharing mechanism for SAGE2 using Content Espresso that makes only minimal modifications to SAGE2. The mechanism is offered based on three assumptions regarding remote collaboration with SAGE2. First, multiple organizations located at various sites are involved in a remote collaboration using SAGE2. Second, the SAGE2 system is installed in each organization. Finally, each organization has multiple users who work with the SAGE2 system using their web browsers.

An overview of the CE-based file sharing mechanism for SAGE2 is follows. First, the files are stored in Content Espresso before sharing. Second, the file reference information is shared among SAGE2 systems by using web technologies. Finally, each SAGE2 system accesses Content Espresso to retrieve files by using shared file reference infor-

Figure 6.1: A typical SAGE2 session; multiple SAGE2 applications are launched on the Display Clients [2].

mation. To minimize modifications to SAGE2 and keep use the existing programming approaches of today's dominant web browsers, the Relay Server is introduced so that SAGE2 systems can access Content Espresso. The Relay Server receives file retrieval and storing requests from the SAGE2 system and relays them to Content Espresso. In addition, the Relay Server serializes concurrent requests and caches the retrieved files to manage and reduce network bandwidth utilization.

A CE-based file sharing mechanism for SAGE2 is designed and implemented. In order to evaluate file sharing performance, *Espresso Image Viewer*, a simple bitmap image viewer using Content Espresso, is developed as an application for SAGE2. The performance of the CE-based file sharing mechanism is evaluated by comparing the file sharing time using the CE-based mechanism with that using the original SAGE2 mechanism. The proposed mechanism can be applied to web-based collaboration systems other than SAGE2 because it is designed and implemented using existing web browser-based technologies.

6.2 Problems of SAGE2

6.2.1 Overview of SAGE2

SAGE2 is a collaborative platform for sharing information on large high-resolution displays with other sites. It enables groups to work together in front screens with the same displays in order to solve problems that require managing a large volume of information in high resolution. SAGE2 is implemented using cloud-based and browser-based technologies in order to enhance data-intensive collocated and remote collaboration. Figure 6.1 offers an example of collaborative work using SAGE2 at a single site.

A SAGE2 system consists primarily of the *SAGE2 Server*, *Display Clients*, *SAGE2 applications*, and *Interaction Clients*. The SAGE2 Server is a customized web server that serves as the core component of the SAGE2 system. The Display Clients present

SAGE application windows on large high-resolution screens. SAGE2 applications provide users with a variety of functions to support collaborative work, while Interaction Clients provide users with the ability to work with SAGE2. Users can launch SAGE2 applications to be viewed with the Display Clients and control those applications using the Interaction Clients via their web browsers.

SAGE2 provides a JavaScript API to create native SAGE2 custom applications; this enables rapid application integration and development in the community that is interested in supporting multi-user collaborative environments. Thus, users are able to develop SAGE2 custom applications with relative ease by using JavaScript and other browser technologies such as jQuery [21].

6.2.2 Remote collaboration on SAGE2

When remote collaboration with SAGE2 involves two organizations, the SAGE2 system is installed in each organization. Figure 6.2 shows an example of a remote collaboration environment using SAGE2. The SAGE2 systems communicate with each other system using HTTP.

The image file sharing procedure of SAGE2 between organization A and organization B is as follows: 1) the image file is stored in the local storage of the SAGE2 Server at organization A; 2) a user launches the default image viewer application and views the image file on the Display Client in organization A using the Interaction Client; 3) that user employs the Interaction Client to share the image file with organization B; 4) the SAGE2 Server in organization B receives the file sharing request and relays it to the organization B Display Client; 5) that Display Client launches its own default image viewer application, and the application retrieves the image file from the SAGE2 Server in the organization A using HTTP; 6) that default image viewer application shows the image file.

6.2.3 Problems in large file sharing

Sharing large files with other organizations using SAGE2 has several problems. First, the SAGE2 system uses HTTP to transmit the files to other SAGE2 systems. HTTP utilizes TCP as its transport layer protocol, and TCP performance declines as the RTT between end nodes increases. Thus, it can take an unacceptably long time to share large files with other organizations, which and devalues the user experience with SAGE2 collaboration.

Second, the available network bandwidth is limited for any organization, no matter the size. When collaborating members of one organization launch their SAGE2 applications and retrieve large files simultaneously, their network will likely become congested, which causes file retrieval performance degradation and again affects productivity.

Finally, the SAGE2 application tends to retrieve the same files again when the application is relaunched. In a collaborative work session, users must sometimes relaunch applications or reload files, which only increases the stress on the organization's network bandwidth utilization.

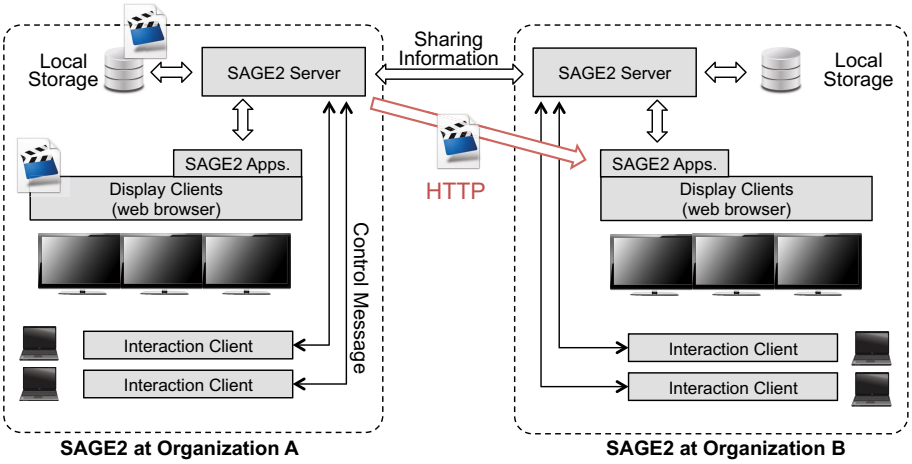


Figure 6.2: An example of a remote collaboration environment using SAGE2 and the image file sharing procedure.

6.3 Design of the proposed mechanism

6.3.1 System overview

This chapter proposes a CE-based file sharing mechanism to improve SAGE2's large file sharing performance. Content Espresso can deliver large files with high throughput regardless of the RTT between SAGE2 systems because it utilizes UDP rather than HTTP as a transport layer protocol. The *Ticket File* is introduced to share file reference information between SAGE2 systems. Ticket files are shared using the existing SAGE2 file sharing mechanism because the value of SAGE2's file sharing interface has been demonstrated.

Suppose that a shared file is already stored in Content Espresso. After sharing the Ticket File, each SAGE2 system retrieves the file from Content Espresso. Since major modifications to the SAGE2 Server are necessary if it is to be made to access Content Espresso, the proposed mechanism has SAGE2 applications access Content Espresso. SAGE2 applications are written in JavaScript and work on Display Clients, one of which is a web browser. Generally, web browsers do not receive UDP packets for security reasons, so a *Relay Server* is introduced to facilitate message exchange between the SAGE2 applications and Content Espresso. The Relay Server receives file retrieval requests from the SAGE2 application using WebSocket, sends those requests to Content Espresso, receives files, and pushes them to the requesting SAGE2 application(s). In addition, the Relay Server serializes concurrent requests from SAGE2 applications and caches the retrieved files to lower network bandwidth utilization. This mechanism can be applied to web-based collaboration systems other than SAGE2 because it is designed and implemented using existing web browser-based technologies.

Figure 6.3 offers an overview of the CE-based file sharing mechanism. First, the file is

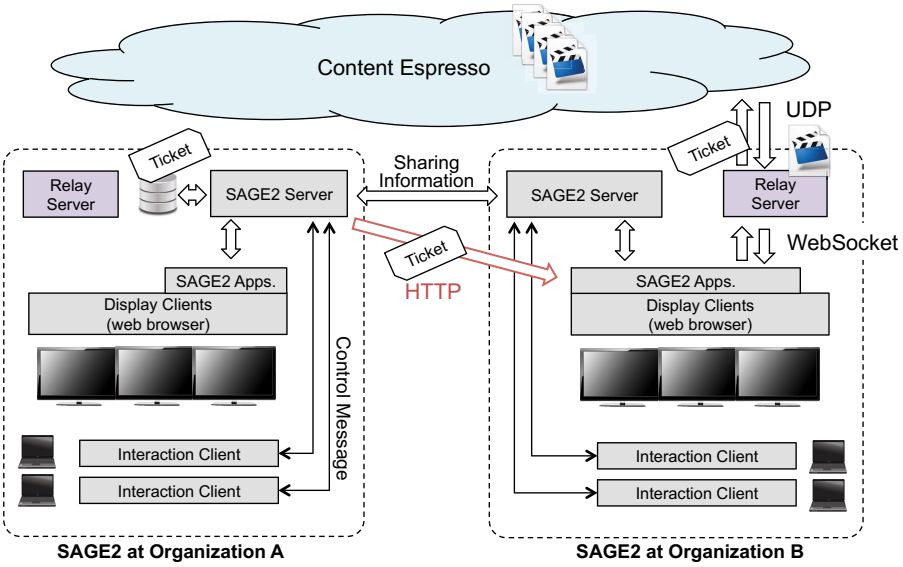


Figure 6.3: Files are stored in Content Espresso; the Ticket File is shared among the users on the existing web-based collaboration system.

stored in Content Espresso before sharing and its Ticket File is stored in the local storage of the SAGE2 Server in organization A. Second, a user operates the Interaction Client to share the Ticket File with the SAGE2 system in organization B. Third, the SAGE2 Server in organization B launches the SAGE2 Application and the SAGE2 Application sends the file retrieval request to the Relay Server in organization B. Finally, the Relay Server retrieves the file from Content Espresso and pushes the file to the requesting SAGE2 Application via WebSocket.

6.3.2 Ticket File format

The proposed mechanism utilizes a Ticket File to share file reference information among SAGE2 systems. The Ticket File is written in XML; its format is the same as the Ticket File used in the multi-domain authentication and authorization system called Yamata-no-Orochi [10, 11].

Figure 6.4 shows an example of the Ticket File format. The Service field describes the service name that issues the Ticket File, the Date field shows the period of validity, the Target field contains the file reference information, and the Ticket File is signed by the service provider to protect the Ticket File from falsification. The Signature field has the signature of the service provider. Since Content Espresso supports authentication and authorization based on Yamata-no-Orochi, Content Espresso can provide file sharing with reliable access control. Thus, the proposed mechanism can support authorized file sharing if necessary.

```
<?xml version="1.0"?>
<OrochiTicket>
  <Service>                                   Service information
    <ServiceName>Content_Espresso</ServiceName>
    <ServiceTicketId>6</ServiceTicketId>
  </Service>
  <Date>                                      Period of validity
    <IssueDate>2016-09-01T06:30:00</IssueDate>
    <NotValidBefore>2016-01-01T00:00:00</NotValidBefore>
    <NotValidAfter>2018-12-31T00:00:00</NotValidAfter>
  </Date>
  <Target>
    <Subjects>
      ..............
    </Subjects>
    <Resources>                               GFID of the target file
      <Resource>00001732000000000000159000000001</Resource>
    </Resources>
    <Actions>
      <Action>Read</Action>
    </Actions>
  </Target>                  Signature of the service provider
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    ..............
  </Signature>
</OrochiTicket>
```

Figure 6.4: An example of the Ticket File format; the Resource field contains the GFID of the target file.

6.3.3 Relay Server

The Relay Server is installed in each organization to provide SAGE2 applications with an access interface to Content Espresso and to control the network bandwidth utilization of the organizations. The Relay Server is composed of three components: *WebSocket Server*, *Request Manager*, and *Content Cache*. Figure 6.5 presents the Relay Server components and the relationship among Content Espresso, Relay Server, and SAGE2 applications.

WebSocket Server

WebSocket Server establishes WebSocket connections and accepts file retrieval and storing requests from SAGE2 applications running on the Display Clients. When the WebSocket Server receives a file retrieval request with a Ticket File from a SAGE2 application, the WebSocket Server checks whether the Content Cache has the requested file. If the file is found, WebSocket Server sends it to the SAGE2 application. Otherwise, it

places the request information in the Request Queue of the Request Manager and waits until the file is delivered to the Content Cache, after which WebSocket Server sends the file to the SAGE2 application. When the WebSocket Server receives a file storing request from a SAGE2 application, it sends that request to Content Espresso using the Content Espresso Client API. After the file is stored, the WebSocket Server receives a Ticket File from the API and sends it on to the SAGE2 application.

Request Manager

The Request Manager has a Request Queue for serializing concurrent file retrieval requests. WebSocket Server enqueues each file retrieval request with a Ticket File in the Request Queue, while the Request Manager dequeues each request from the Request Queue and sends it to Content Espresso using the Content Espresso Client API. After a file is retrieved, the Request Manager places the retrieved file in the Content Cache and notifies the WebSocket Server that file retrieval has been completed.

Request Manager manages the bandwidth utilization for file retrieval. Content Espresso uses UDP to transmit chunks from Chunk Servers to clients. Excessive bandwidth utilization causes packet loss and file retrieval performance degradation. Thus, each client must determine the best chunk retrieval rate. The Request Manager controls the file retrieval interval so as not to exceed the configured bandwidth.

Content Cache

The Content Cache keeps the data, GFID, status, and last access time of retrieved files unless the available space for caching runs out. Retrieved files are likely to be requested multiple times during collaborative work. If the Content Cache has the requested file, that can reduce file retrieval time and bandwidth utilization. The size of a specific cache space can be configured by a system administrator. When the cache space runs out, the cache item that has not been used in the longest time is discarded so that a new cache item can be inserted.

The Content Cache entry has two types of status: VALID and INVALID. The cache entry is created by WebSocket Server when it enqueues the file retrieval request in the Task Queue. A VALID status means that the cache entry has the correct file data, while an INVALID status means that the cache entry does not have the file data because the Request Manager did not finish retrieving the file.

6.4 Implementation

6.4.1 Relay Server

The Relay Server is implemented using the Content Espresso Client API in C++. Figure 6.6 presents the implementation of the Relay Server. The Relay Server is composed of a Content Cache, a Task Queue, and two types of threads: WebSocket threads and Request

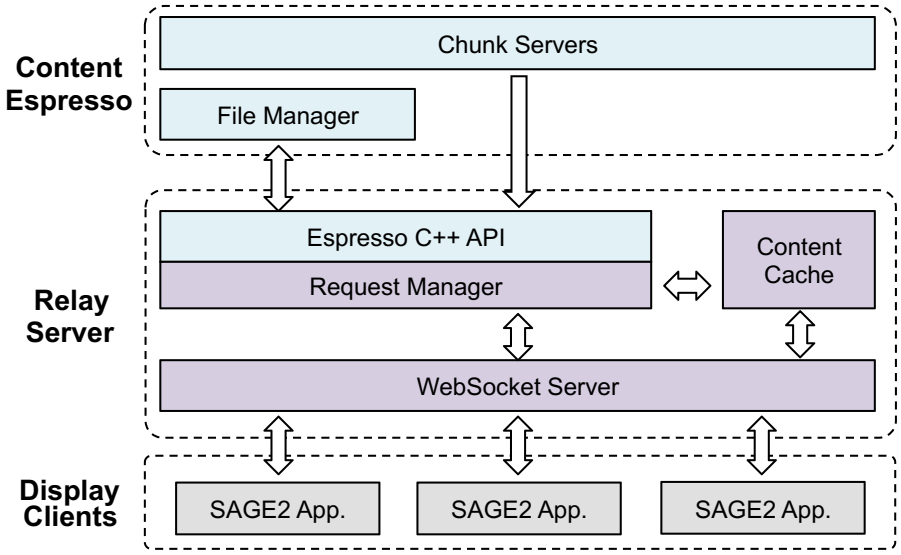


Figure 6.5: WebSocket Server, Request Manager, and Content Cache comprise the Relay Server.

threads. A WebSocket thread is created for each WebSocket connection and receives the file retrieval request from the web browser. Request threads are implemented using a thread pool approach, dequeue file retrieval requests from the Task Queue, and send them to Content Espresso using the Content Espresso Client API.

The Relay Server's file retrieval procedure is as follows: 1) A SAGE2 application establishes a WebSocket connection with the WebSocket Server; 2) the application sends a Ticket File to the WebSocket Server; 3) the WebSocket Server parses the Ticket File and acquires the file ID of the requested file; 4) the WebSocket Server checks whether the Content Cache already has the requested file; 5) if the Content Cache has a cache entry for the requested file and the cache status is VALID, the WebSocket Server sends the file to the Web browser using the WebSocket connection; 6) if the Content Cache has a cache entry for the requested file but the cache status is INVALID, the WebSocket Server waits until the status changes to VALID; 7) if the Content Cache does not have a cache entry for the requested file, the WebSocket Server makes a cache entry with an INVALID status in the Content Cache, enqueues the file retrieval request in the Task Queue, and waits until file retrieval has been completed; 8) the Request Manager checks the Task Queue, and if the request exists in the Task Queue, the Request Manager dequeues the request and sends the file retrieval request to Content Espresso; 9) the Request Manager finishes the file retrieval, adds the retrieved file to the cache entry, changes that status to VALID, and notifies the WebSocket Server; 10) the WebSocket Server sends the retrieved file to the web browser for use in the relevant SAGE2 application.

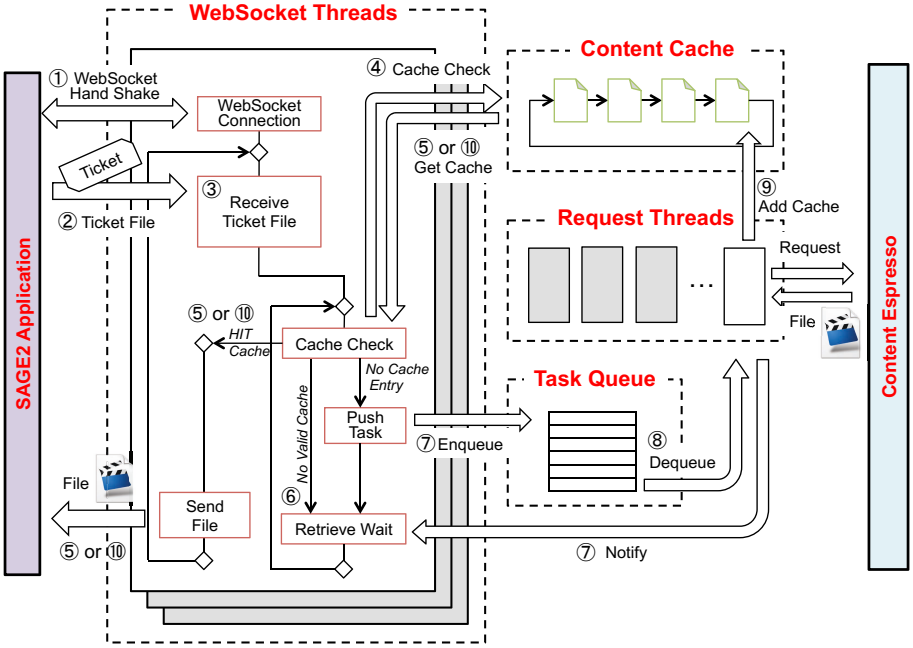


Figure 6.6: The Relay Server is composed of the Content Cache, the Task Queue, and two types of threads, the WebSocket thread and the Request thread.

6.4.2 Espresso Image Viewer

A simple bitmap image viewer called *Espresso Image Viewer* is also implemented using the SAGE2 JavaScript API. The basic function of the Espresso Image Viewer is to display bitmap files stored in Content Espresso on the SAGE2 Display Client. When a Display Client user employs the Interaction Client to open a Ticket File, the Espresso Image Viewer is launched on that user's display. Then, the Espresso Image Viewer sends the Ticket File to the Relay Server to retrieve the target file, acquires the target bitmap file, and displays it. On the other hand, when a client employs Espresso Image Viewer to have the Interaction Client to store a bitmap file, it sends the file to the Relay Server to store it to Content Espresso and receives a Ticket File. The Espresso Image Viewer then stores the Ticket File in the local SAGE2 Server storage.

6.5 Performance evaluation

6.5.1 Evaluation environment

In order to evaluate the performance of a CE-based file sharing mechanism on SAGE2, 16 physical servers were set up in Japan for Content Espresso and two physical client machines in Hawaii for SAGE2. Figure 6.7 shows the experimental environment, while the specifications of each physical machine are shown in Table 6.1.

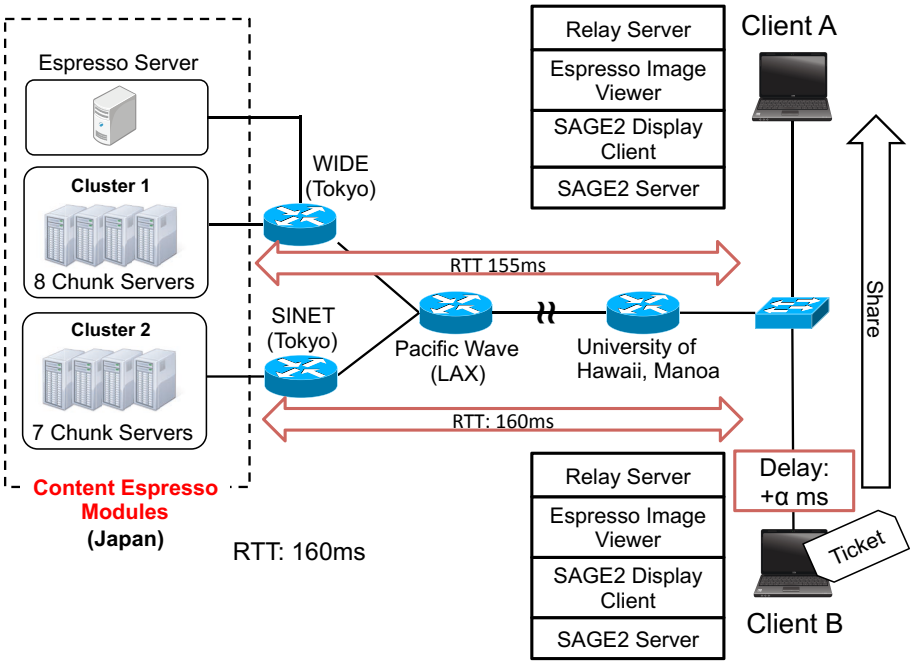


Figure 6.7: Evaluation environment.

Content Espresso is composed of five modules: File Manager, Storage Allocator, Chunk Generator, Cluster Head, and Chunk Server. 15 physical machines are utilized as Chunk Servers and divided into two Chunk Server Clusters (Cluster 1 and Cluster 2). One physical machine at each cluster was also utilized as a Cluster Head. The File Manager, Storage Allocator, and Chunk Generator were installed on a remaining single physical server called the Espresso Server. SAGE2 is composed of the SAGE2 Server, Display Clients, Interaction Clients, and SAGE2 applications. A SAGE2 Server and Display Client were installed on the client machines. In addition, a Relay Server was installed on the client machines.

The Chunk Servers in Cluster 1 and an Espresso server were connected to a router with a 1Gbps link. The Chunk Servers in Cluster 2 were connected to another router with a 1Gbps link. The Cluster 1 network was connected to a router at the University of Hawaii at Manoa via the SINET network, while the Cluster 2 network was connected to a router at the University of Hawaii at Manoa via the WIDE network. Both networks were connected to a router in Los Angeles before being connected to the Hawaiian location. The average RTTs between the clients and Chunk Servers in Cluster 1 and Cluster 2 were 155 ms and 160ms respectively. A network delay could be added by using a Network Link Conditioner with the computers relevant to client B.

Table 6.1: Specifications of physical machines.

Modules	OS	CPU
Client A	CentOS 6	Corei7 (3.40GHz)
Client B	MacOS 10.8	Corei7 (3.40GHz)
Chunk Server	Ubuntu 12.04	Pentium (2.80GHz)
Espresso Server	CentOS 6	Corei7 (3.40GHz)

6.5.2 File sharing time comparison

This evaluation compares file sharing time with changing RTTs between client A and client B using three types of image files: an uncompressed Full HD (1920 x 1080 pixel, 6.2MB) image file, an uncompressed UHD (3840 x 2160 pixel, 24.9MB) image file, and an uncompressed 8K (7680 x 4320 pixel, 99.5MB) image file. The additional RTT ranged from 1 ms to 300 ms, to take actual Internet circumstances into consideration.

Figures 6.8 to 6.10 show the results of file sharing time comparisons between the existing SAGE2 and proposed CE-based file sharing mechanisms on SAGE2, using a 6.2 MB Full HD image file, a 24.9 MB UHD image file, and a 99.5 MB 8K image file respectively. These figures show that, with the original SAGE2 file sharing mechanism, file sharing time increased rapidly as the RTT between clients becomes larger. This is because that mechanism uses HTTP to transmit large image files, and HTTP utilizes TCP as a transport layer protocol; the performance of TCP worsens as RTTs between senders and recipients become larger.

On the other hand, the file sharing time in the CE-based file sharing mechanism increased gradually, because that mechanism only uses HTTP to transmit the Ticket File and retrieves large image files using Content Espresso. The Ticket File is vastly smaller than an image file and Content Espresso utilizes UDP for retrieving the image file. Thus, the CE-based file sharing mechanism does not contribute significantly to performance degradation.

In addition, these figures show the CE-based file sharing mechanism is faster than the original SAGE2 file sharing mechanism when the RTT between the clients is above 10 ms; the difference in sharing time becomes significantly large when the RTT is 300 ms. The CE-based file sharing mechanism takes about two, three, and eight seconds for the full HD, UHD image, and 8K images, respectively, while the existing SAGE2 file sharing mechanism takes about 17, 50, and 200 seconds respectively.

6.5.3 File sharing throughput

The evaluation above confirmed that CE-based file sharing improves file sharing performance when the RTT between the clients is longer than 10ms by demonstrating the throughput of that mechanism with a 6.2 MB Full HD image file, a 24.9 MB UHD image file, and a 99.5 MB 8K image file. Figure 6.11 shows the throughput of the CE-based file

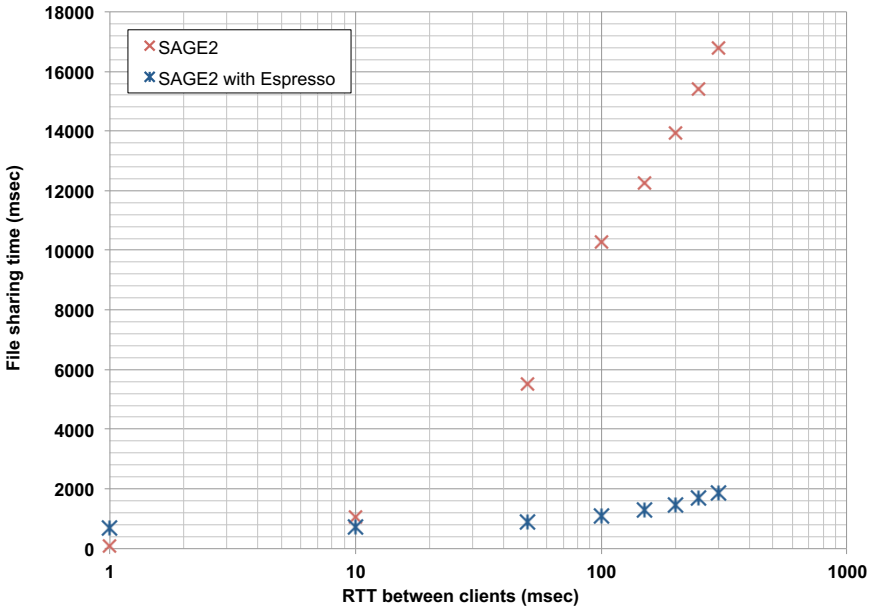


Figure 6.8: File sharing time comparison when a 2.4MB Full HD image file is used.

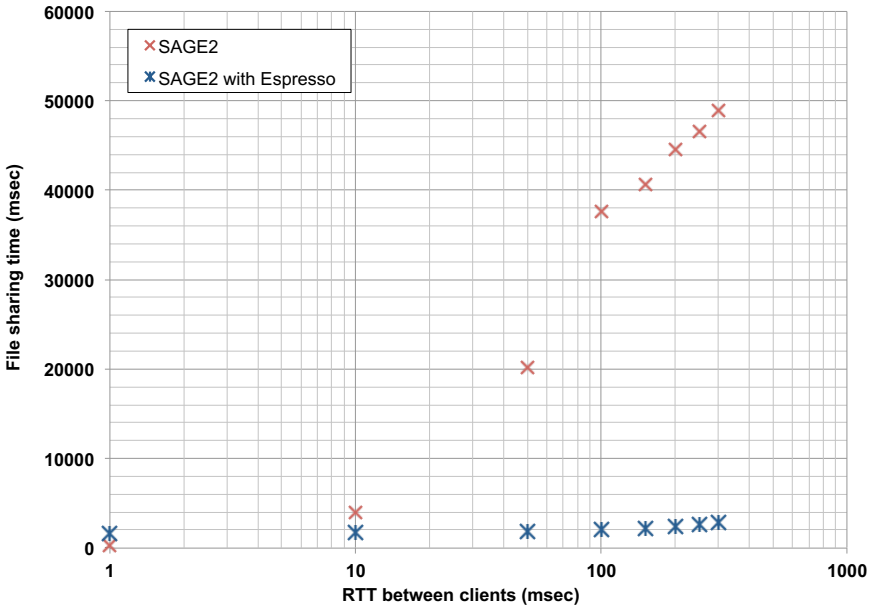


Figure 6.9: File sharing time comparison when a 6MB UHD image file is used.

sharing mechanism. The results show that throughput was better as the file size became larger because the file retrieval overhead in Content Espresso becomes comparatively

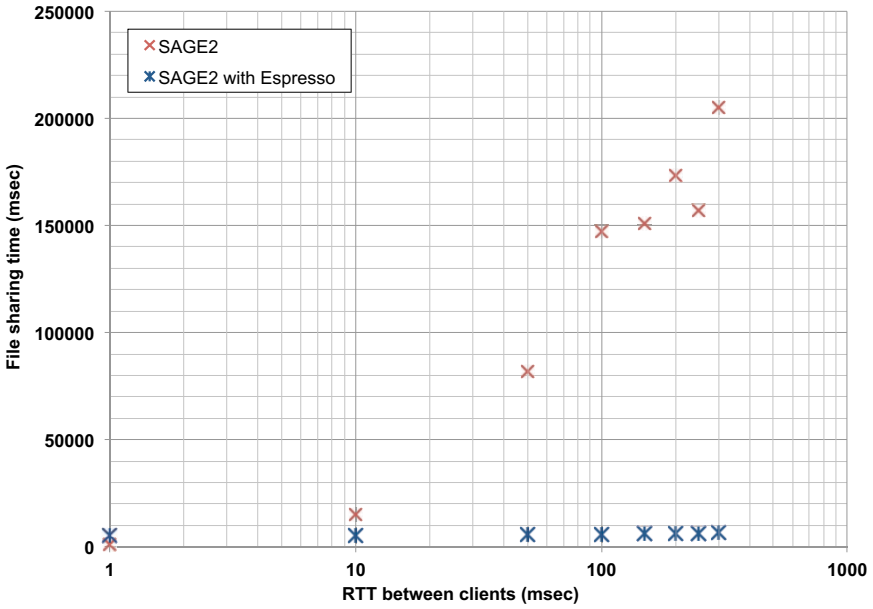


Figure 6.10: File sharing time comparison when a 10MB 8K image file is used.

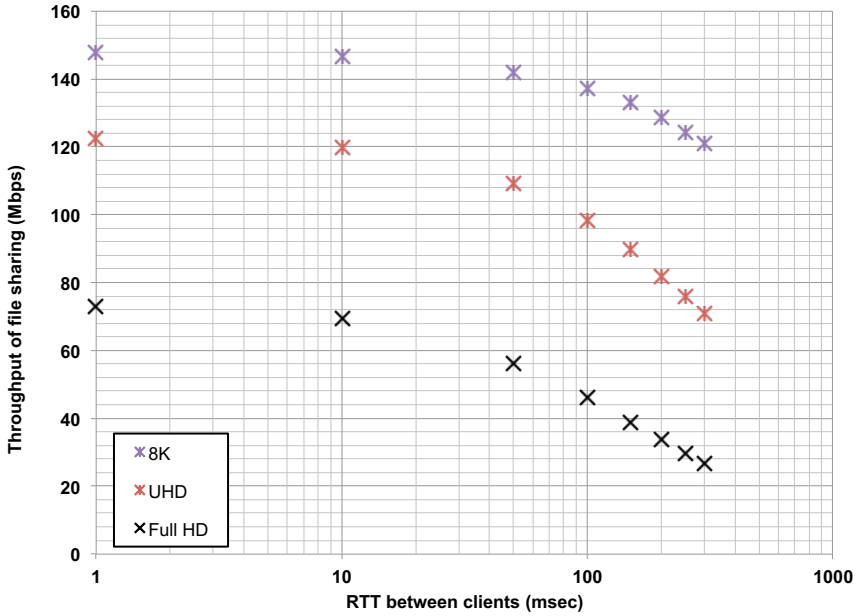


Figure 6.11: Throughput of CE-based file sharing on SAGE2.

small when file sizes are larger. Thus, the proposed mechanism is suitable for large file sharing.

6.6 Summary

This chapter has proposed a global high throughput large file sharing mechanism for web-based collaboration systems that uses Content Espresso. SAGE2 was selected as a typically data-intensive collaboration system. SAGE2 shares information on large high-resolution displays with other sites using web browser-based technologies. The CE-based file sharing mechanism was designed and implemented by introducing the Ticket File and Relay Server and integrating the mechanism into SAGE2, without requiring significant modifications to that software.

In order to evaluate the proposed file sharing mechanism, the Espresso Image Viewer was implemented as a SAGE2 Application. This chapter has evaluated the proposed mechanism by comparing file sharing times between the existing SAGE2-based file sharing mechanism and its CE-based file counterpart using Full HD, UHD, and 8K uncompressed image files. The results of the evaluation have confirmed that the proposed mechanism offers better performance than the existing SAGE2 mechanism when the RTT between the sites is longer than 10 ms, with any type of file. The CE-based file sharing mechanism was shown to improve large file sharing performance in remote collaboration environments. The proposed mechanism can be applied to web-based collaboration systems other than SAGE2.

Chapter 7

Related Work

7.1 Overview

This chapter shows work related to Content Espresso. DRIP is a fundamental mechanism of Content Espresso; it uses UDP to transmit the stored chunks and FEC to recover lost chunks. Therefore, this chapter first discusses work related to transport layer protocols to confirm why Content Espresso uses UDP-based chunk transmission. Then, this chapter reviews and compares systems similar to Content Espresso from the viewpoint of distributed storage systems. It then reviews FEC algorithms by comparing several useful examples as application-layer FECs. Finally, studies of packet loss pattern models on the Internet are reviewed, because Content Espresso was evaluated by emulating the packet loss actually found on the Internet in Chapter 4.

7.2 Transport layer protocol

Content Espresso adopts the file storage and retrieval mechanism shown in Chapter 2. That mechanism uses UDP as a transport layer protocol to transmit chunks from Chunk Servers to Client. This section shows related work that aims at high throughput data transmission in a global environment. The transport layer protocols for high throughput data transmission can be classified into TCP-based or TCP-like protocols and UDP-based protocols.

7.2.1 TCP-based protocol

In order to achieve high throughput data transmission, many approaches have been proposed. Modified TCP approaches such as FAST TCP [22], HighSpeed TCP [23], and Scalable TCP [24] all reduce performance degradation in the delayed network and/or lossy network environment by modifying the algorithms of the congestion control mechanism and/or the rate-controlling mechanism.

Parallel TCP approaches such as MPTC [25], SCTP [26], PSockets [27], CAVERNsoft [28], and GridFTP [29] use multiple TCP connections simultaneously and aggregate them to achieve high-bandwidth data transmission. MPTC and SCTP are transport layer implementations that improve data transmission throughput and resiliency. However, when all paths share the same bottleneck link, the performance of MPTC and SCTP actually becomes worse than standard TCP. PSockets, GridFTP, and CAVERNsoft offer an application-level TCP connection aggregation approach. Although these application-level parallel TCPs do achieve high throughput data transmission compared to standard TCP, their performance degrades when network delays are long and/or packet loss rates are large.

7.2.2 UDP-based protocol

In order to avoid the performance degradation due to network delays and packet losses, UDP-based protocols [30, 31, 32, 33, 34, 35, 36] have been proposed. This subsection discusses representative UDP-based protocols.

RBUDP [31] aims to keep the network pipe as full as possible during bulk data transfer and avoid per-packet interactions. The sender establishes a TCP connection to send and receive a signaling message before sending the actual data to the recipient(s). After that, the sender sends the data using UDP. After data are sent, recipients send acknowledgement messages to the sender. If some packets are lost, the sender retransmits the lost packets. Since RBUDP adopts retransmission to achieve reliable data transfer, it is important to estimate the available bandwidth correctly by using existing tools like iperf [37], netperf [38], and nuttcp [39]. By contrast, Content Espresso does not retransmit lost chunks; it recovers them by using FEC. Therefore, Content Espresso does not have to estimate the available bandwidth correctly.

UDT [30] is a reliable UDP-based application-level data transport protocol for distributed data-intensive applications over high-speed WANs. UDT uses packet-based sequencing and timer-based selective acknowledge. Unlike RBUDP, UDT has its own rate control and bandwidth estimation mechanisms. In addition, the acknowledgement message is sent as soon a packet loss event is detected and the lost packets are retransmitted. Therefore, the performance of UDT becomes worse when the network packet loss rate becomes higher.

QUIC (Quick UDP Internet Connections) [36] was designed at Google and aims at improving the performance of HTTP-based data transmission in WANs. QUIC achieves high throughput data transmission regardless of network delays between senders and recipients by supporting multiplexed connections, providing a secure connection similar to TLS/SSL, reducing latency between senders and recipients, and estimating the available bandwidth. Although QUIC does provide high throughput data transmission in WANs, it also retransmits lost packets, which causes performance degradation.

7.3 Distributed storage system

Many distributed storage systems have been proposed and used for data storing and sharing. They usually split a given file into small data chunks and store them to multiple storage servers or devices with redundant data to achieve high availability, high reliability, and high throughput, much like Content Espresso. Distributed storage systems can be classified by redundancy technique, IO unit, and the number of storage providers utilized.

7.3.1 Redundancy technique

There are three main redundancy techniques to achieve high reliability and availability; using replication, employing FEC, and hybrids of the two. For example, GFS [40], HDFS [41], and Wheel FS [42] use replication, while RobuSTore [43] and Cloud-RAID [44] use FEC. Although replication is widely employed because its implementation is simple, FEC requires users to calculate parities and recover lost data.

On the other hand, FEC requires less storage space than replication to achieve the same availability and reliability levels. OceanStore [45] and Pond [46] use both replication and FEC. They use FEC to achieve reliable preservation by employing highly reliable storage servers, which can be expensive, and replication to achieve high availability. Since Content Espresso aims at low storage cost, Content Espresso uses FEC for achieving both high availability and reliability.

7.3.2 IO unit

Distributed storage systems can also be classified by IO unit; block, object, and file. With distributed block storage systems such as OceanStore [45], HDFS [41], and iSCSI RAID [47], files are split into evenly sized blocks of data and stored on multiple storage devices. Distributed block storage systems manage file metadata in metadata servers but usually do not have the metadata of each block of every file. Thus, users access specific file blocks by using API (typically POSIX-like API) with file identifiers and offset of file data.

With distributed object storage systems such as Ceph [48], Lustre [49], and Panasas [50], files are split into various-sized data objects and stored in dedicated object storage clusters. Distributed object storage manages the metadata of each object on metadata servers and allows users to access that data. Although object storage has th advantage of partial updating, the performance of metadata servers worsens as the number of objects grows larger.

The IO unit of Content Espresso is a file; it does not manage metadata of chunks but of files. Thus, Content Espresso reduces the number of metadata access events to some extent. However, this situation can also occur using distributed object storage when a file consists of only a single object. In order to clarify the technological differences

between Content Espresso and distributed object storage systems, Content Espresso, Ceph, Gfarm, and GFS are compared in Table 7.1.

Table 7.1: Comparison of Content Espresso, Ceph, Gfarm, and GFS.

	Espresso	**Ceph**	**Gfarm**	**GFS**
IO Unit	File	Object	File	Chunk
Unit Size	Dynamic	Dynamic	Dynamic	64MB
Data Storage	Chunk Server	RADOS	FNS	GN
Data Location	Simple Striping	CRUSH	DHT	Avoid the same rack of the replications.
Redundancy Tech.	FEC	Replication	Replication	Replication
Redundancy Level	Determined by each owner of file	Determined by storage administrator	Determined by storage administrator	Typically three replicas
Metadata Management	Owner Domain Based Approach	Dynamic Subtree Partitioning	MDS	Centralized Approach

7.3.3 Number of data centers

The number of storage providers is one of the most important aspects of distributed storage systems. This thesis defines distributed storage systems that use either a single provider or multiple storage providers as the single storage provider model or the multiple storage providers model respectively. It is assumed that RAID [6] and iSCSI RAID [47] are used in LANs and that Spanner [51] uses a single data center to avoid high data management cost. These technologies are classified into the single storage provider model.

On the other hand, it is assumed that SPANStore [52], SafeStore [53], DEPSKY [54], and Cloud-RAID [44] use multiple storage providers. SPANStore replicates files to multiple cheap storage providers to achieve low cost and low latency and satisfy the fault tolerance requirements of each application. This technique is quite similar to CDNs such as Akamai [55], Limelight Networks [56], Mirror Image [57], and Level 3 [58], all of which cache frequently accessed content files in edge servers to provide low-latency access to those key files. Cloud-RAID splits files into small data chunks and stores them on multiple storage providers; it utilizes FEC to achieve high availability and reliability, which is similar to Content Espresso's approach. The performance evaluation of Cloud-RAID shows that using multiple storage providers can improve the IO throughput, but that overall performance is dependent on the throughput performance of specific storage providers. These technologies, including Content Espresso, are classified in the multiple storage providers model.

7.3.4 Example of distributed storage system

This subsection examines each of the representative distributed storage systems in terms of the criteria shown above.

Ceph

Ceph [48] is designed to achieve high throughput by using commodity hardware similar to Content Espresso. Ceph stores object data in object storage devices (OSDs) called RADOS, while Content Espresso stores chunks in the ext4 file system of Chunk Servers. As discussed previously, metadata server performance in distributed object storage systems worsens as the number of objects in a file grows larger. In order to avoid this problem, Ceph has a highly reliable centralized metadata server cluster and adopts Dynamic Subtree Partitioning [59]; however, preparing many highly reliable servers carries heavy costs. While Content Espresso stores chunks by striping to Chunk Servers, Ceph utilizes CRUSH [60], which reduces object data movement and achieves scalability by appending new OSDs. When new Chunk Servers are added in Content Espresso, it is not necessary to move stored chunks to the new Chunk Servers because Content Espresso stores chunks evenly across all Chunk Servers. Since both Ceph and Content Espresso are designed to achieve high throughput by using commodity hardware, this thesis dis-

cusses the performance differences between Ceph and Content Espresso. [61] shows that Ceph achieves 4Gbps read performance when using TCP tuning, four servers, and 4MB IO, thanks to the open-source RADOS Bench tool. This result shows that Ceph has better performance than Content Espresso, at least within certain parameters. However, this evaluation does not take into account either a truly global environment or the cost of storage servers. Since the evaluation above utilizes TCP, the read performance would be worse in a global environment. In addition, that evaluation uses high-performance, 10-HDD servers intended for high-performance computing (HPC), which adds significant costs to the organization seeking to work collaboratively across great distances.

Gfarm

The Gfarm Grid file system [62] is a global distributed file system for sharing data and supporting distributed data-intensive computing. Storage of the Gfarm file system uses a federation or collection of local file systems of computer nodes or commodity PCs. Gfarm is composed of multiple file system nodes (FNS) a Metadata Server (MDS), and a Gfarm client. This client is the library used to access the MDS and FNS from user applications. The MDS manages file system metadata, file open status, file system node status, global user accounts, and group membership information. The FNS is the storage server and composed by Gfarm client, I/O Server, and local file system. Gfarm adopts the file replication to ensure reliability and achieve high performance of data access file replicas are managed by the central server to support close-to-open.

GFS

The Google File System (GFS) [40] is a scalable distributed file system for large distributed data-incentive applications. GFS consists of a single master, which manages the metadata of stored chunks and multiple chunkservers that store the chunk data. The two system assumptions of GFS that are similar to those of Content Espresso are as follows: each system is built from very inexpensive commodity hardware and each stores a large number of large files. However, GFS is assumed to be employed with the MapReduce storage system [63], which is a framework for processing large data in large clusters. Therefore, GFS focuses on the improvement of simultaneous read/write performance. For example, GFP adopts a 64MB static-size chunk to reduce interaction with the master, which does indeed improve read/write performance. GFS makes replications of chunks to ensure system availability and reliability. Since it is assumed that GFS will be deployed in a single data center, it is necessary to consider replica placement. GFS typically has hundreds of chunkservers spread across many machine racks; GFS spreads chunk replicas across racks to ensure data reliability.

7.3.5 Cost analysis

Low cost storage is one of the core Content Espresso goals. This subsection discusses the cost of Content Espresso in comparison with Ceph. Basically, Content Espresso entails storage preparation costs, storage utilization costs, and networking costs. The storage and retrieval mechanism of Content Espresso utilizes FEC rather than replication because FEC can achieve the same level of availability and reliability at lower storage utilization cost.

Content Espresso allows users to select the amount of redundancy for each file on their own, which can help reduce storage utilization costs in comparison to systems that adopt a fixed amount of redundancy such as RAID. As to storage preparation cost, Content Espresso has an advantage because it is assumed that the system uses commodity hardware. For this thesis, Content Espresso was actually installed in a deployment that used 72 Chunk Servers, each of which costs less than US$300. Thus, the total storage preparation cost is about $21,600, which is much cheaper than how Ceph is configured in [61]. On the other hand, the Content Espresso consumes more network bandwidth than a replication-based approach because the system sends all chunks, including parity chunks, to the client, which boosts networking costs. In order to mitigate this issue, the overall amount of parity chunks sent for a given should be adjustable based on estimating network conditions, but this remains a future goal.

7.4 Forward error correction

FEC is the key technology of Content Espresso because it utilizes FEC in the application layer to recover lost chunks caused by storage server failure and packet loss during transmission. In general, FEC in the application layer is called Application Layer FEC (AL-FEC); a variety of FEC algorithms are used for AL-FEC, such as Reed-Solomon [64], low-density parity check (LDPC) [65], LDGM [8], and Raptor Code [66]. Reed-Solomon is used for many distributed storage systems, including RAID [6], OceanStore [45], and RobuSTore [43]. However, decoding speed worsens as FEC block sizes grow larger. [67] has shown that LDPC, LDGM, and Raptor Code demonstrate better performance when the FEC block size is large. Turbo coding [68] also has better performance in terms of error recovery ratio. [69] compares the performance and complexity of LDPC and Turbo coding. The evaluation confirms that LDPC has a significantly lower complexity than Turbo coding and LDPC recovers lost data faster than Turbo coding. Since Turbo coding has a fixed number of iterations in the decoder, it is default to reduce the decoding time. On the other hand, LDPC has potential for improving decoding time by multiplexing decoding process. Since it is assumed that Content Espresso use large FEC blocks, LDGM has been selected for this implementation. Content Espresso is designed to allow for users to select their preferred FEC algorithm, but actually implementing multiple FEC algorithm options in Content Espresso is a future goal.

7.5 Packet loss pattern

Chapter 4 evaluates the performance of Content Espresso by emulating a packet loss environment using the tc command. Currently, three packet loss models are implemented with the tc command: the random loss model, the four-state Markov model, and the Gilbert-Elliott loss model (which itself has special cases known as the Gilbert loss model, the simple Gilbert loss model, and the Bernoulli loss model). This section discusses studies of packet loss patterns on the Internet.

Gilbert [70] and Elliott [71] introduced a two-state Markov approach called the Gilbert-Elliott model, which is widely used for emulating packet loss on the Internet. The Gilbert-Elliott model is outlined in Figure 7.1, where **G** is the good state and **B** is the bad state. k is the probability that the packet is transmitted while the system is in the good state, h is the probability that the packet is transmitted while the system is in the bad state, p is the loss probability, $1-r$ is the burst duration, and $1-h$ is the loss density.

The `tc` command adds packet losses according to the Gilbert-Elliott loss model and its special cases (the Gilbert loss model, Simple Gilbert loss model, and Bernoulli loss model) when the user selects their loss model. The Gilbert model is the Gilbert-Elliott model where $k = 1$. The simple Gilbert model is the Gilbert model where $h = 0$. The Bernoulli model is the Gilbert model where $1 - r = p$. [72] found a simple Gilbert model ($k = 1, h = 0$) useful for describing the characteristics of packet losses in Internet connections and for deriving an error model for Internet video transmissions on top, as lost packets will affect the perceived quality of the video transmission. Therefore, this thesis selects the simple Gilbert model as a packet loss model.

Other than the two-state Markov model, [73] proposed a four-state Markov model. Figure 7.2 depicts the four-state Markov model: state 1 means the packet was received successfully; state 2 means the packet was received within a burst; state 3 means the packet was lost within a burst; and state 4 means that the packet was lost within a gap. This model is widely used to emulate the IEEE 802.11 channel. Since this thesis focuses on packet loss on the Internet, the four-state Markov model was not selected for evaluating the system.

7.6 Summary

This chapter has reviewed several studies related to Content Espresso. First, the transport layer protocol was reviewed. Since the performance of TCP degrades when the RTT between the file sender and the file recipient becomes large or the packet loss rate increases, a TCP-based protocol is not appropriate for transmitting files in a global environment. Several studies have tried to solve the problem by using parallel TCP or modified TCP. However, they cannot avoid performance degradation. Thus, Content Espresso selects UDP as the transport layer protocol. Several studies have focused on UDP's transmission of files in a global environment. However, most examples retransmit lost packets because UDP is an unreliable protocol, which causes performance degradation. Content

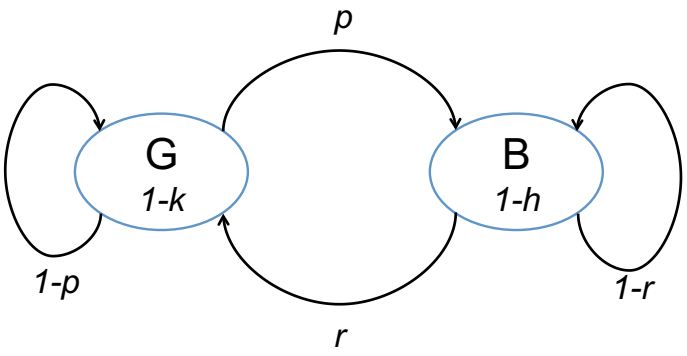


Figure 7.1: Gilbert-Elliott model.

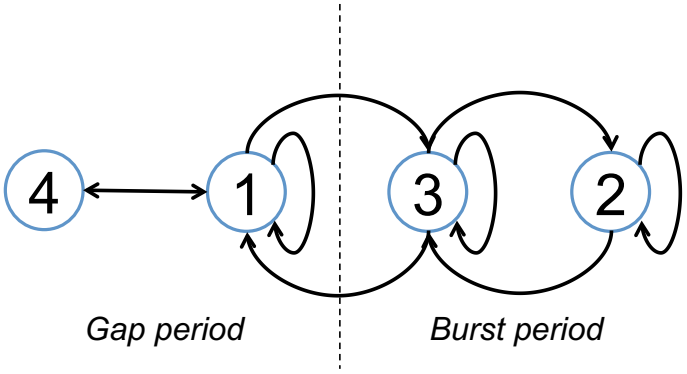


Figure 7.2: Four-state Markov model.

Espresso uses FEC to avoid having to retransmit lost chunks.

Second, the distributed storage systems were reviewed. The features of Content Espresso are similar to distributed storage systems because Content Espresso uses a metadata server and multiple storage servers. This review revealed the differences between Content Espresso and other systems from the viewpoint of redundancy technique, IO unit, and the number of data centers. Third, the features of several FEC algorithms were reviewed. Content Espresso adopts LDGM coding as its FEC algorithm because its performance is better than the other algorithms with large block sizes. Finally, packet loss patterns on the Internet were reviewed. This thesis evaluated Content Espresso's file retrieval and storage performance in a global environment in Chapter 4 by emulating network packet loss rates and discussing it in comparison with several models of packet loss on the Internet.

Chapter 8

Conclusion

8.1 Summary of this thesis

Large content file sharing over the Internet among people and organizations all over the world has now become widespread with the rapid growth of multimedia devices, networking technologies, and cloud technologies. For example, video production companies usually share source video files with post-production companies to post-production processes such as video editing, thus enabling low cost content creation. However, challenges remain in sharing a large number of large files with people with at low cost and high speed on the Internet.

This thesis has introduced and detailed Content Espresso, which is a high throughput large file sharing system for a global environment. Content Espresso uses four key techniques: first, UDP was chosen to avoid performance degradation when the RTT or packet loss rate between file senders and recipients becomes long and large, respectively. In order to evaluate Content Espresso, an experimental environment was set up with 79 physical machines, including 72 inexpensive storage servers. This thesis evaluated file metadata access performance, file storage and retrieval performance, FEC block size, and system availability by emulating global environments with the `tc` command. The results confirm that Content Espresso is capable of dealing with 15,000 requests per second, achieving 1Gbps for file storage, and achieving more than 3Gbps for file retrieval. File storage and retrieval performance are not significantly affected by network conditions. Thus, it can be concluded that Content Espresso demonstrates the capability to serve as a high throughput file sharing system on a global scale.

This thesis also demonstrated two Content Espresso applications: a network-oriented uncompressed UHD video playback system called Demitasse and an approach to improving the file sharing performance of SAGE2. Demitasse stores video component files to Content Espresso and the relationships among those files to Catalogue System, which is a distributed graph database. The Demitasse Client retrieves the relationships among the video component files, obtains their GFIDs, retrieves the appropriate file from Content Espresso, and plays back the file. This thesis measured the video playback perfor-

mance of Demitasse by using uncompressed Full HD and UHD video frame files. The evaluation confirmed that Demitasse achieves playback of uncompressed UHD 30fps video with a 7.19Gbps throughput involving FEC, with less than 0.1% frame loss by file retrieval requests. Demitasse contributed to improving and evaluating the file retrieval performance of Content Espresso.

SAGE2 is a web-based collaboration system that shares information on large high-resolution displays with other sites, enabling people in multiple locations to work together and see the same displays. Since SAGE2 uses HTTP to share files, it takes a long time to share those files in a global environment. This thesis integrates Content Espresso into SAGE2 as a file sharing system to provide a high throughput file sharing function. SAGE2 works on web-based technologies and communicates with other modules by using HTTP. However, Content Espresso does not communicate in HTTP. In order to translate the protocol between SAGE2 and Content Espresso, the Relay Server is introduced. This thesis measured file sharing time by emulating a global environment. The results confirmed that Content Espresso improves the file sharing performance of SAGE2 when the RTT to remote sites is greater than 10ms. This study contributes to showing that Content Espresso can be applied to web-based applications.

8.2 Future work

This thesis mainly introduces Content Espresso. Since Content Espresso uses UDP without any congestion control mechanism to deliver files with high throughput, its traffic probably disturbs other TCP-based traffic such as HTTP traffic. It is important to solve this problem if Content Espresso is used in the actual Internet. The author believes that this problem should be solved by two approaches. The first approach is to use network-virtualization technologies. Network-virtualization technologies split the physical network into several virtual network slices and each slice is assigned to each application. These technologies separate Content Espresso traffic from other application traffics. The second approach is to implement an automatic chunk delivering rate control mechanism in Content Espresso. Although the first approach can solve that Content Espresso's traffic disturbs other applications' traffic, it cannot avoid disturbing other Content Espresso's traffic. In order to avoid it, Content Espresso should have the mechanism to determine the chunk retrieval rate automatically taking network occupation into consideration. This is future work of this study.

Finally, future plans of Content Espresso and its applications are mentioned. Content Espresso shows that the one possibility to use UDP and FEC to store and delivering files in the Internet. The author believes that Content Espresso is worth to use for various industries that keep a large number of large files. In order to deploy Content Espresso and its applications in the society, The author believes that Content Espresso and its applications should be released as open source software and managed in a community. The author starts considering concrete release and management plans of them from now on.

Bibliography

[1] Daisuke Ando, Fumio Teraoka, and Kunitake Kaneko. Storage and Retrieval Mechanism for Large Files using Global Distributed Servers. *IEICE Transactions on Communications (Japanese Edition)*, Vol. 97, No. 10, pp. 861–872, 2014.

[2] T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. Mateevitsi, L. Renambot, L. Long, A. Johnson, and J. Leigh. SAGE2: A new approach for data intensive collaboration using Scalable Resolution Shared Displays. In *Proceedings of International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pp. 177–186, Oct. 2014.

[3] Jacob Poushter. Smartphone ownership and Internet usage continues to climb in emerging economies. *Pew Research Center*, 2016.

[4] Dropbox. `https://www.dropbox.com/`.

[5] Google Drive. `https://www.google.com/drive/`.

[6] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *SIGMOD Record*, Vol. 17, No. 3, pp. 109–116, Jun. 1988.

[7] Motion Picture Association. Theatrical Market Statistics 2015. Apr. 2016.

[8] D. J. C. MacKay and R. M. Neal. Near Shannon limit performance of low density parity check codes. *Electronics Letters*, Vol. 33, No. 6, pp. 457–458, Mar. 1997.

[9] Sae-Young Chung, G. D. Forney, T. J. Richardson, and R. Urbanke. On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit. *IEEE Communications Letters*, Vol. 5, No. 2, pp. 58–60, Feb. 2001.

[10] Yuki Atsuya, Kunitake Kaneko, and Fumio Teraoka. Yamata-no-Orochi: an Authentication and Authorization Infrastructure for Internet Services. *IPSJ Journal (Japanese Edition)*, Vol. 55, No. 2, pp. 849–864, Feb. 2014.

[11] Kei Mikami, Daisuke Ando, Kunitake Kaneko, and Fumio Teraoka. Verification of a Multi-Domain Authentication and Authorization Infrastructure Yamata-no-Orochi. In *Proceedings of the 11th International Conference on Future Internet Technologies (CFI 2016)*, pp. 69–75, 2016.

[12] MariaDB Foundation. Maria DB. `https:/mariadb.org/`.

[13] NTT America. Global Virtual Link SLA. `https://www.us.ntt.net/support/sla/global-virtual-link.cfm`.

[14] NTT Europe. SLA of Global IP Network. `http://www.eu.ntt.com/jp/services/network/ip-network-transit/sla-of-global-ip-network.html`.

[15] Dan Kegel. The C10K problem. `http://www.kegel.com/c10k.html`, 2006.

[16] Luigi Rizzo and Matteo Landi. Netmap: Memory mapped access to network devices. *SIGCOMM Computer Communication Review*, Vol. 41, No. 4, pp. 422–423, Aug. 2011.

[17] DPDK. DPDK (DATA PLANE DEVELOPMENT KIT). `http://dpdk.org/`.

[18] OpenGL. `https://www.opengl.org/`.

[19] GLFW. `http://www.glfw.org/`.

[20] Google Apps. `https://apps.google.com/`.

[21] jQuery. `https://jquery.com`.

[22] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (TON 2006)*, Vol. 14, No. 6, pp. 1246–1259, Dec. 2006.

[23] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), Dec. 2003.

[24] Tom Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *SIGCOMM Computer Communication Review*, Vol. 33, No. 2, pp. 83–91, Apr. 2003.

[25] Sébastien Barré, Christoph Paasch, and Olivier Bonaventure. Multipath TCP: from theory to practice. *NETWORKING 2011*, pp. 444–457, 2011.

[26] R Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sep. 2007.

[27] Harimath Sivakumar, Stuart Bailey, and Robert L Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC 2000)*, pp. 38–38, 2000.

[28] Kyoung S. Park, Yong J. Cho, Naveen K. Krishnaprasad, Chris Scharver, Michael J. Lewis, Jason Leigh, and Andrew E. Johnson. CAVERNsoft G2: A Toolkit for High Performance Tele-immersive Collaboration. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 2000)*, pp. 8–15, 2000.

[29] Nicolas Kourtellis, Lydia Prieto, Adriana Iamnitchi, Gustavo Zarrate, and Dan Fraser. Data Transfers in the Grid: Workload Analysis of Globus GridFTP. In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing (DADC 2008)*, pp. 29–38, 2008.

[30] Yunhong Gu and Robert L Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks*, Vol. 51, No. 7, pp. 1777–1799, 2007.

[31] E. He, J. Leigh, O. Yu, and T.A. DeFanti. Reliable Blast UDP : predictable high performance bulk data transfer. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, pp. 317–324, 2002.

[32] Kunitake Kaneko and Naohisa Ohta. Design and implementation of live image file feeding to dome theaters. *Future Generation Computer Systems*, Vol. 27, No. 7, pp. 944–951, 2011.

[33] Yunhong Gu and Robert Grossman. SABUL: A transport protocol for grid computing. *Journal of Grid Computing*, Vol. 1, No. 4, pp. 377–386, 2003.

[34] Xuan Zheng, Anant Padmanath Mudambi, and Malathi Veeraraghavan. Frtp: Fixed rate transport protocol–a modified version of sabul for end-to-end circuits. In *Proceedings of Broadnets*, 2004.

[35] Qishi Wu and Nageswara SV Rao. Protocol for high-speed data transport over dedicated channels. In *Proceedings of Third International Workshop on Protocols for Long-Distance Networks (PFLDnet 2005)*, 2005.

[36] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. QUIC: A UDP-based secure and reliable transport for HTTP/2. *IETF, draft-tsvwg-quic-protocol-02*, 2016.

[37] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects`, 2005.

[38] Rick Jones, et al. NetPerf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

[39] nuttcp. `http://www.nuttcp.net/`.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pp. 29–43, 2003.

[41] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pp. 1–10, 2010.

[42] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2009)*, Boston, MA, Apr. 2009.

[43] Huaxia Xia and Andrew A. Chien. RobuSTore: A Distributed Storage Architecture with Robust and High Performance. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC 2007)*, pp. 1–11, 2007.

[44] M. Schnjakin, D. Korsch, M. Schoenberg, and C. Meinel. Implementation of a secure and reliable storage above the untrusted clouds. In *Proceedings of the 8th International Conference on Computer Science Education (ICCSE 2013)*, pp. 347–353, Apr. 2013.

[45] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 190–201, 2000.

[46] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pp. 1–14, 2003.

[47] Xubin He, Praveen Beedanagari, and Dan Zhou. Performance evaluation of distributed iSCSI RAID. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2003)*, 2003.

[48] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pp. 307–320, 2006.

[49] S Donovan, G Huizenga, AJ Hutton, CC Ross, MK Petersen, and P Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium 2003*, 2003.

[50] Clusters Brent Welch, Brent Welch, and Garth Gibson. Managing Scalability in Object Storage Systems for HPC Linux. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 433–445, 2004.

[51] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, Vol. 31, No. 3, pp. 1–22, Aug. 2013.

[52] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 2013)*, pp. 292–308, 2013.

[53] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. SafeStore: A Durable and Practical Storage System. In *Proceedings of USENIX Annual Technical Conference*, pp. 129–142, 2007.

[54] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage (TOS)*, Vol. 9, No. 4, pp. 12:1–12:33, Nov. 2013.

[55] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Operating Systems Review*, Vol. 44, No. 3, pp. 2–19, Aug. 2010.

[56] Limelight Networks. `https://www.limelight.com/`.

[57] Mirror Image. `http://www.mirror-image.com/`.

[58] Level 3. `http://www.level3.com/en/products/content-delivery-network/`.

[59] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC 2004)*, p. 4, 2004.

[60] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC 2006)*. ACM, 2006.

[61] Feiyi Wang, Mark Nelson, Sarp Oral, Scott Atchley, Sage Weil, Bradley W Settlemyer, Blake Caldwell, and Jason Hill. Performance and Scalability Evaluation of the Ceph Parallel File System. In *Proceedings of the 8th Parallel Data Storage Workshop*, pp. 14–19, 2013.

[62] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm Grid File System. *New Generation Computing*, Vol. 28, No. 3, pp. 257–275, 2010.

[63] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113, 2008.

[64] Irving S Reed and Gustave Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, Vol. 8, No. 2, pp. 300–304, 1960.

[65] Vincent Roca, Christoph Neumann, and David Furodet. Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes. Technical report, 2008.

[66] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, Vol. 52, No. 6, pp. 2551–2567, Jun. 2006.

[67] R. Montalban Gutierrez and G. Seco-Granados. Efficiency comparison of LDPC-LDGM and Raptor codes for PL-FEC with very large block sizes. In *Proceedings of Wireless Telecommunications Symposium 2009*, pp. 1–6, Apr. 2009.

[68] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Proceedings of IEEE International Conference on Communications (ICC 1993)*, Vol. 2, pp. 1064–1070, May 1993.

[69] Kjetil Fagervik and Arne Sjøthun Larssen. Performance and complexity comparison of low density parity check codes and turbo codes. In *Proceedings of Norwegian Signal Processing Symposium (NORSIG 2003)*, pp. 2–4, 2003.

[70] E. N. Gilbert. Capacity of a burst-noise channel. *The Bell System Technical Journal*, Vol. 39, No. 5, pp. 1253–1265, Sep. 1960.

[71] E. O. Elliott. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal*, Vol. 42, No. 5, pp. 1977–1997, Sep. 1963.

[72] Bernd Girod, Klaus W. Stuhlmueller, M. Link, and U. Horn. Packet-loss-resilient Internet video streaming. In *Proceedings of SPIE 3653*, pp. 833–844, Dec. 1998.

[73] Jeff McDougall, Jeevan Joseph John, Yi Yu, and Scott L Miller. An improved channel model for mobile and ad-hoc network simulations. In *Proceedings of Communications, Internet, and Information Technology (CIIT 2004)*, pp. 352–357, 2004.