Title
Toward petabyte digital content transfer and preservation over optical networks

Permalink
https://escholarship.org/uc/item/7pc7v3r2

Author
Liu, Shaofeng

Publication Date
2010

Peer reviewed|Thesis/dissertation
Toward Petabyte Digital Content Transfer and Preservation Over Optical Networks

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science by Shaofeng Liu

Committee in charge:

Jurgen Schulze, Chair
Thomas A. DeFanti
Alin Deutsch
Jeanne Ferrante
Miller Puckette
Larry Smarr
Alex C. Snoeren

2010
The dissertation of Shaofeng Liu is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2010
DEDICATION

To my family.
The best way to predict the future is to create it!

—Abraham Lincoln
# TABLE OF CONTENTS

Signature Page ................................................................. iii  
Dedication ........................................................................ iv  
Epigraph ............................................................................. v  
Table of Contents .............................................................. vi  
List of Figures ....................................................................... ix  
List of Tables ....................................................................... xi  
Acknowledgements ............................................................. xii  
Vita and Publications .......................................................... xiii  
Abstract of the Dissertation ................................................ xiv  

Chapter 1  Introduction ......................................................... 1  
1.1 Using Optical Networks for Parallel Transfer of Visual Digital Content ......................................................... 3  
1.2 Using Optical Networks to Facilitate Distributed Digital Archiving ................................................................. 6  
1.3 Contributions ................................................................... 7  

Chapter 2  Related Works ......................................................... 9  
2.1 Related Research Communities ........................................ 9  
2.1.1 Scientific Visualization ............................................... 9  
2.1.2 CineGrid ................................................................ 10  
2.1.3 Data Grid ................................................................ 10  
2.2 Data Transfer Techniques and Challenges ....................... 10  
2.2.1 Packet Switching Networks ...................................... 11  
2.2.2 Network Protocols ................................................. 12  
2.2.3 Packet Loss Recovery ........................................... 13  
2.2.4 Forward Error Correction ....................................... 17  
2.3 Will Parallel Data Transfer Be The Trend? ...................... 19  

Chapter 3  Parallel Data Transfer Using Cross-Stream Coding ................................................................. 20  
3.1 Introduction .................................................................. 20  
3.2 Parallel Data Transfer Challenges .................................. 21  
3.3 Cross-Stream Coding ................................................... 25  
3.3.1 Simple Cross-Stream Coding Mechanisms .................. 26
LIST OF FIGURES

Figure 1.1: The trend of the Internet traffics: 1990-2010 Sources: Cisco estimates based on CAIDA publications, Andrew Odlyzko ........ 2
Figure 1.2: The latest map of GLIF............................................. 3
Figure 1.3: Parallel display examples........................................ 5
Figure 2.1: Representation forms of fountain code..................... 18
Figure 3.1: Parallel data loss pattern..................................... 23
Figure 3.2: An analogical example of water pouring.................. 24
Figure 3.3: Sample matrices. The portion in the big circle marked on M2 identified a regular fountain code matrix; the chained four small circles identified a pipe packet in M3. S=4, N=4, N'=5, N"=6 X-axis: code packets Y[4][6] Y-axis: original data packets V[4][4] CS code packets (marked in red): Y_{13}, Y_{25}, Y_{32}, Y_{44} ......................... 30
Figure 3.4: Decoding procedure.............................................. 31
Figure 3.5: Code matrix data structure................................. 33
Figure 3.6: Raptor code overhead........................................... 39
Figure 3.7: Protocol overhead can be calculated from this figure. The original number of data packet is 4340 including pre-code, Raptor Code overhead can be read from Figure 3.6, which is about 7.5% for N=1024, and from this figure, the overall recovery percentage becomes 1 when N"=1230, so protocol overhead is 1230 \times \frac{4340 \times 1.075}{4340 \times 1.075} - 1 = 5.45%. ................................................................. 40
Figure 3.8: These lines show the function of frame recovery percentage to the number of issued packet to receivers, and the lower figure is the zoom-in of the top-right part of the upper figure. (X-axis: Number of code packets; Y-axis: Percentage of recovered original packets) ........ 41
Figure 3.9: CSTP vs. pure FEC (X-axis: data stream ID from 1 to 10; Y-axis: packet recovery percentage) ........................................ 43
Figure 3.10: CSTP vs. RBUDP in parallel data transfer (X-axis: Number of Parallel Data Streams; Y-axis: Overall Data Transfer Speed in MB/s) ........ 44
Figure 4.1: Effectiveness of a CS code................................. 54
Figure 4.2: A useless CS code.............................................. 55
Figure 4.3: A useless CS code.............................................. 56
Figure 4.4: A useful CS code.............................................. 57
Figure 4.5: Instant Effectiveness of a CS code...................... 58
Figure 4.6: Incremental decoding cycle using CS codes............... 59
Figure 4.7: Incremental Effectiveness.................................... 62
Figure 4.8: Incremental Effectiveness curves.......................... 63
Figure 4.9: Accumulative Effectiveness curves.......................... 65
Figure 4.10: An example of $f(P)$ .......................................................... 68
Figure 4.11: Small portion of packet loss can be self-recovered. .............. 69
Figure 4.12: The red portion is beyond the recoverability of self-decoding. ... 70
Figure 4.13: The next stream is $S(P + \frac{1}{3})$ ........................................... 71
Figure 4.14: $f(P)$ .................................................................................. 74
Figure 4.15: A optimal solution derived from L1 theorem .......................... 76
Figure 4.16: $L2(P)$ vs. OBJ(P) ($\delta = 5\%$) .............................................. 81
Figure 4.17: The $L2(P)$ curve of a L1 solution ($\bar{\delta} = 15.1\%$) ............ 82
Figure 4.18: $L2(P)$ vs. OBJ(P) ($\delta = 3\%$) .............................................. 83
Figure 4.19: Communication Cost vs. Space Cost ...................................... 84
Figure 4.20: CS code efficiency ................................................................. 87
Figure 4.21: Using L3 to find the best feasible solution ($\gamma = 20\%$) ........ 91
Figure 4.22: Using L3 to find the best feasible solution ($\gamma = 11.5\%$) ..... 92
Figure 4.23: Linear loss pattern (X-axis: P; Y-axis: f(P)) ............................ 93
Figure 4.24: Recovery of linear loss pattern .............................................. 93
Figure 4.25: Stepped loss pattern (X-axis: P; Y-axis: f(P)) .......................... 94
Figure 4.26: Recovery of stepped loss pattern ........................................... 95
Figure 4.27: Emulated loss pattern on FWGrid ........................................ 96
Figure 4.28: An optimal solution to recover loss pattern on FWGrid .......... 96
Figure 4.29: The actual emulation results (The pink line draws the number of final available codes in each of the streams.) ......................... 97

Figure 5.1: Map of the CX sites ................................................................. 103
Figure 5.2: The CX architecture ............................................................... 109
Figure 5.3: The CX ingest workflow (the portion of the workflow with a dark background is automated using CX middleware) ....................... 112
Figure 5.4: CX access interfaces .............................................................. 116
Figure 5.5: Parallel transfer throughput from San Diego to Tokyo (X-axis: number of parallel transfers; Y-axis: throughput in Mbytes/sec) ....... 117
LIST OF TABLES

Table 3.1: Definitions in CSTP .................................................. 28
Table 3.2: Decoding speed of the Raptor Code ................................ 39
Table 4.1: Synchronization speed .................................................. 48
Table 5.1: CX nodes and their capacity ......................................... 102
Table 5.2: The CX file transfer performance ................................. 115
ACKNOWLEDGEMENTS

Thanks to Dr. Henri Casanova and Dr. Jeanne Ferrante for advising and funding me in the first two years of my Ph. D study. Thanks to Dr. Thomas A. DeFanti, Dr. Larry Smarr and Dr. Jurgen P. Schulze for advising me and supporting my thesis work in the last four years. Special thanks to Cisco Systems, Inc. for providing the majority of funding for my Graduate Student Researcher position in the last four years. Without these help, I couldnot have successfully completed the Ph. D program in UCSD.

Thanks to CineGrid community. CineGrid has motivated a big part of my research. Thanks to Academy of Motion Picture Arts and Sciences for participating in the related CineGrid projects in my dissertation. Thanks to iRODS team for working together on related CineGrid projects. Thanks to California Institute for Telecommunication and Information Technology(Calit2) for providing the research facilities.

Chapter 3, in part, has been published, in Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage. Jurgen Schulze and Thomas A. DeFanti co-authored the paper. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, has been submitted for publication of the material as it may appear in the journal of Future Generation Computer Systems, 2011. Jurgen P. Schulze and Thomas A. DeFanti co-authored this paper. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, has been submitted for publication of the material as it may appear in the journal of Future Generation Computer Systems, 2011. Jurgen P. Schulze, Laurin Herr, Jeffrey D. Weekley, Bing Zhu, Natalie V. Osdol, Dana Plepys, Mike Wan co-authored this paper. The dissertation author was the primary investigator and author of this paper.

Thanks to my family in China. They have given me so much courage and wisdom towards the challenges of life and study. Thanks to Jing Zhu for helping prepare for thesis proposal and dissertation.
VITA AND PUBLICATIONS

1999
B. S. in Computer Science, Tsinghua University, P.R.China

1999-2001
M. S. in Computer Science, Tsinghua University, P.R.China

2004-2010
Graduate Student Researcher, in Department of Computer Science and Engineering, University of California, San Diego

2010
Ph. D. in Computer Science, University of California, San Diego


ABSTRACT OF THE DISSERTATION

Toward Petabyte Digital Content Transfer and Preservation Over Optical Networks

by

Shaofeng Liu

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Jurgen Schulze, Chair

Optical networks have become the central elements in many scientific applications. And the emerging of those data intensive applications also brings about new challenges to the approaches on how to efficiently use the abundant bandwidth provided by optical networks.

From some of the applications, we observe a parallel data access model, and are thus motivated to study a new approach to use parallel data streams to better feed the parallel data access model. Parallelization of data streams leads to new challenges which are not sufficiently addressed in previous related works. The new approach we propose in this thesis improves existing approaches through tightly coupling data streams using Cross-Stream coding. It can effectively resolve unevenly distributed code packet loss
at the receiving side to eliminate the necessity of frequently sending acknowledgements back to the senders requesting missing data. We further prove that the Cross-Stream coding approach is scalable, and give out practical algorithms to find optimal solutions for large number of parallel streams, for any given objective Cross-Stream packet loss pattern. We experimented with our Cross-Stream coding approach using computer clusters.

We also observe, from other applications, that optical networks can improve traditional digital archiving methods, and will include our preliminary research of using optical networks to facilitate a workflow based digital archiving system, named CineGrid Exchange. The CineGrid Exchange has been practically online for about a year, storing and managing 20 Tera bytes digital contents.

In summary, both the requirements of large scale network applications and the available bandwidth of optical networks increase faster than the development of middleware between them. And there exist significant challenges to alleviate the difference. The research in this thesis is motivated by these challenges, and explores possible directions towards future petascale data transfer and preservation.
Chapter 1

Introduction

The Internet is never fast enough. We, as users of the Internet, have witnessed the speed of the Internet grow from 56Kbps via a modem, to 10/100Mbps via ADSL or cable networks installed in hundreds of millions of homes in the world. Users’ utilization of the Internet has also evolved. Besides browsing simple webpages, more and more people are watching high resolution videos online. From the recent report [Odl10], shown in Figure 1.1, there is a clear trend about how the Internet is used vastly differently as new network applications appear. The development of network technologies just barely meet the requirements of new network applications, and in many cases, applications have to adapt to the available network bandwidth. Although new applications have leveraged the continuous development of the Internet infrastructure, they put great pressure onto the future Internet technology as the same time. That’s perhaps why we never feel the Internet is sufficiently fast.

On the other hand, although the speed of the Internet keeps growing, we cannot take it for granted that the trend will continue naturally without significant technological improvements towards future generations of the Internet applications. From the perspective of network researchers experimenting with high-speed private optical networks built and shared by research organizations, although the speed of these networks have grown much faster than the Internet, i.e. 1Gbps to 10Gbps networks have become standard configurations, the exploration of how to fully use those high speed infrastructures is still at an early stage. Tremendous efforts are made to develop applications to leverage the availability of optical networks. For example, OptIPuter [Sma09] is a re-
Figure 1.1: The trend of the Internet traffics: 1990-2010

Sources: Cisco estimates based on CAIDA publications, Andrew Odlyzko

Recent research project whose mission is "to enable collaborating scientists to interactively explore massive amounts of previously uncorrelated data", through the "central architectural element of OptIPuter: optical networking". Furthermore, research institutes contribute and share networking resources to form a larger community which can better promote remote collaboration via these resources. GLIF [GLI10], for example, is "an international virtual organization that promotes the paradigm of lambda networking", which has grown to include many research networks, as shown in Figure 1.2. People are enthusiastic about fast networks, and care about how to utilize future faster networks to efficiently support emerging new applications.

As time goes on, many applications that used to appear in science fiction stories become true due to fast networks, and gradually, the communication modules running on the lower level of the applications to transfer data are becoming increasingly critical. There is a large demand for improving the performance of those modules. We cannot change the trend that future applications will have higher networking requirements, i.e., speed, latency; instead, we should think whether the potential of current network resources had been fully leveraged. To be specific, the current communication modules used by those applications need to be improved to transfer data more efficiently. We believe these network communication modules should adapt to network applications, but
not the opposite. Since applications may have different characteristics, communication modules need to take them into consideration. And communication protocols may vary between different types of network applications.

The research in this thesis was initially motivated by one specific type of application emerged in the recent few years: large scale digital content transfer, streaming and archiving.

1.1 Using Optical Networks for Parallel Transfer of Visual Digital Content

There has been massive research on optical networks in recent years. As one example, CineGrid[ Cin10] has led a trend of using high bandwidth networks to transfer very high quality digital content for real-time movie showings and digital preservation. Although many successful demonstrations of 4K video [SKF+09, LH05] streaming have been performed over fiber networks, we foresee the challenges in designing new protocols which scale to applications demanding higher bandwidth and higher resolution digital content in the future. Today, ultra-high-resolution displays have become standard infrastructure in scientific research. These displays are typically built by tiling an array of standard LCD displays into a display wall, driven by a PC cluster. Figures 1.3a and 1.3b show different types of display walls: the HIPerSpace uses sixty 30’ LCD dis-
plays to form a display wall with more than 200 million pixels, the StarCAVE [DDS+09] uses 34 high-definition projectors to construct a 3D virtual space where people can work with 3D virtual reality objects. Meanwhile, high-speed research optical networks make it possible for scientists to use these ultra-high resolution displays over long distances for such scientific applications as very-high-definition video streaming or conferencing, real-time data visualization generated by remote scientific instruments, etc. As a perfect example of the combination of display walls and a high-speed network, the OptIPuter research project, funded by the US National Science Foundation, constructed 1Gbps-10Gbps optical network infrastructure as well as middleware to facilitate the interactive access to remote gigabyte to terabyte visualization data objects and bring them to a visual interface, the OptIPortal [DLR+09].

However, scaling up visualization devices from a single PC with a single display to a cluster of PCs with a cluster of displays has brought up challenges on how to feed data to these display devices. Those challenges are based on the limitations of traditional single data source communication model, and create the need for a new communication model with multiple-to-multiple end points, which achieves very-high-bandwidth parallel data transfer/streaming between cluster-based display devices while still appearing as a point-to-point communication protocol to the users.

Traditional data transport protocols have limitations which prevent their use for such multi-endpoint connections. The popular Transport Control Protocol (TCP) is slow on long distance networks because of its window-based congestion control mechanism. Alternatives like RBUDP [HLYD02], UDT [GG07], and LambdaStream [VLEH06] are more recently developed UDP-based protocols focusing on high-speed file transfer or real-time data streaming between two end nodes. These protocols are point-to-point rate-based, supporting one sender and one receiver. The sender controls the sending rate to minimize packets loss. And lost UDP packets are resent. RBUDP, for instance, uses a bitmap to maintain a list of lost UDP packets and do a multi-round communication to recover lost packets, which usually takes 2-5 round trips time (RTT) to retrieve a GB file correctly. LambdaStream detects packet loss based on the gap between the receiving time of two consecutive UDP packets, and if that gap is bigger than expected, the sender will reduce the sending rate accordingly. None of these protocols scale to a
(a) HIPerSpace, one of the world’s largest display walls in Calit2, UC San Diego, has 286,720,000 effective pixels.

(b) The StarCAVE, a third-generation CAVE and virtual reality OptIPortal in Calit2, UC San Diego, has 68,000,000 pixels.

**Figure 1.3**: Parallel display examples
multiple-to-multiple communication model in a straightforward way. Existing parallel data streaming protocols do not sufficiently address the synchronization issue of multiple parallel data streams caused by unevenly-distributed packet loss. These multiple-to-multiple communication protocols [JRJ+06] tend to synchronize senders and receivers by adding a synchronization module to senders and/or receivers. This synchronization module waits until all senders/receivers have the next frame of data, and then broadcasts the command of sending/displaying to all senders/receivers. In these cases, the behaviors of individual data streams will affect the overall performance, for instance burst packet loss of one data stream can slow down all other data streams.

In this thesis, we will explore new methods to transfer data in parallel.

1.2 Using Optical Networks to Facilitate Distributed Digital Archiving

In addition to data transfer methods, digital storage and archiving is also critical. In this thesis, we will show our preliminary research about how to use optical network and open source software to build a scalable distributed storage platform for future digital content preservation. The article “The Digital Dilemma” [SotAoMPAA08] pointed out that the rapidly increasing use of digital technology in the acquisition, post-production and distribution of media content not only brings significant benefits to the motion picture industry, as well as other stakeholders in the media ecosystems, but also raises serious issues on how to manage the large amount of resulting digital content efficiently over long periods of time. Given that a single version of a movie created in ultra high quality can fill tens of terabytes of digital storage space, current technologies for preserving large amounts of data seem far behind the industry’s needs. A unique feature of digital media content is its mobility. For instance, starting from its creation in the life cycle of a movie, digital media will need to be moved from one system to another many times for post-production, cinema distribution, long term archiving, and on-demand retrieval. Therefore, how to manage and transfer digital content efficiently becomes increasingly important. The traditional method of delivering film cans by courier was initially adapted to deliver hard drives and data tapes. But with the increasing
volume of data transfers required for modern media productions which are themselves increasingly distributed around the world, physical delivery and physical preservation of digital media assets no longer satisfy the industry’s requirements. On the other hand, large amounts of long distance fiber optical cable have been installed during the past decade, which now makes dedicated 1Gbps-10Gbps fiber connections more affordable. This trend is making it increasingly practical to transfer large digital media files between remote sites. Furthermore, a distributed storage model has the potential of unifying resources around the world to form a petabyte scale distributed storage platform for media exchange and preservation.

In this thesis, we also introduce our efforts of using optical networks to facilitate digital archiving.

1.3 Contributions

There are many challenges to achieve future large-scale data transfer, streaming and archiving. In response to the challenges, the research in this thesis makes the following contributions:

1. The major contribution of this work is the invention of a new scalable parallel data transfer/storage method via a robust Cross-Stream Coding mechanism. Cross-Stream coding couples many parallel streams to transfer data in parallel, and in particular, unevenly distributed data loss within a threshold can be recovered at the receiving side.

2. We have shown that the overhead of this method is only determined by the parallel data loss pattern, and will not increase with the number of parallel streams. The approach can be scaled up to a theoretically unlimited number of parallel streams.

3. The theorems we have proven evidently explain the conditions of valid Cross-Stream coding schemes, and can yield an optimal solution in linear time with realistic assumptions. The theorems can also be used to predict the decoding result for any real code packet loss pattern.
4. In addition, to address the digital preservation issues, we designed CineGrid Exchange, a distributed storage architecture, which uses optical network and middleware to implement a workflow based digital content preservation system.

The thesis is organized as follows. We introduce related works in Chapter 2. In Chapter 3 we present the new approach about using Cross-Stream Coding to implement a parallel data transfer protocol called CSTP, and in Chapter 4, we prove the scalability of Cross-Stream Coding theoretically. The CineGrid Exchange(CX) architecture is discussed in Chapter 5. Chapter 6 summarizes contributions and suggest future works.
Chapter 2

Related Works

2.1 Related Research Communities

The research described in this thesis borders on a few related disciplines, which are going to be listed below.

2.1.1 Scientific Visualization

Large scale display devices with very high resolution exist in many research institutes. In the article "Visualizing Science: The OptIPuter Project" [PDQL], we learned that visualizing large amount of data on a display wall with hundreds of millions of pixels has become more than a convenient tool which provides an exciting viewing experience: it is essentially needed to interpret the enormous amounts of data produced by large-scale instrumentation, experiments and stimulations. In practice, large scale display devices are typically tiled display walls, driven by a cluster of computers. Each computer in the cluster is only responsible for a small portion of the visualized data, and all computers are synchronized to refresh their images at the same pace.

However, feeding data to these huge display walls is very challenging. The dilemma is: a single data source cannot feed data fast enough to satisfy the data consumption ability of a gigantic display device; feeding with parallel data sources is hard to implement and may cause synchronization issues and significantly degrade the viewing experience.
2.1.2 CineGrid

A similar dilemma exists in high resolution video streaming. CineGrid is a community that pioneered the research of how to archive, share and stream very high definition digital content over optical networks. The mission of CineGrid is to “To build an interdisciplinary community that is focused on the research, development, and demonstration of networked collaborative tools to enable the production, use, preservation, and exchange of very-high-quality digital media over photonic networks.”.

In CineGrid, streaming very high resolution video on top of gigabit optical networks provides very impressive viewing experiences. The main method to achieve that, however, is an ad-hoc solution: original 4K clips are compressed into a 500Mbps data stream at the sender and are decompressed at the receiver in real time using dedicated hardware JPEG2000 codecs. This solution does not scale when a higher resolution image format is required in the future.

2.1.3 Data Grid

Data Grid is related to our research as it provides us with tools for large scale data archiving. We use the iRODS [RRM10] data grid to store and manage many of the high resolution video clips held by CineGrid community. Data grid provides a seamless way to access files in a universal distributed storage system, and also facilitate us to define workflows for content ingest, distribution and archiving.

However, tremendous challenges remain in current data grid implementations. In the case of CineGrid, we replicate every piece of content at least three times and distribute them into three distant locations connected by 10Gbps optical networks. We found that synchronizing files between those locations was extremely slow. A new file transfer paradigm is needed to boost the performance of data grid as well.

2.2 Data Transfer Techniques and Challenges

Large scale data transfer is at the core of many of the applications studied in the above listed related disciplines. In this section, we briefly introduce related data transfer
techniques as a basis for our further discuss in the following chapters.

2.2.1 Packet Switching Networks

The Internet is a packet switching network. Packet switching is “a digital networking communications method that groups all transmitted data-regardless of content, type, or structure-into suitably-sized blocks” [pac], called packets. Packet switching features the delivery of variable-bit-rate data streams (sequences of packets) over a shared network. When traversing network adapters, switches, routers and other network nodes, packets are queued, resulting in variable delay and throughput depending on the traffic load in the network. When we browse web pages, watch video from Youtube, chat over instant messaging, and use point-to-point (P2P) software to share files, all the data are transfered over the Internet using packet switching techniques.

The research networks we work with are implemented using packet switching as well, for example the OptIPuter project, so named for its use of Optical networking, Internet Protocol, computer storage, processing and visualization technologies, is a visionary infrastructure which tightly couple computational resources over parallel optical networks using the IP communication mechanism. For these reasons, we are going to assume that we need to base our work on packet switching networks.

Viewing from the level of network applications, we basically only use networks to perform two tasks: send and receive data from the network. The data we want to send is wrapped in packets by the computer’s operating system and sent off by the computer’s network interface card (NIC). It travels through network cables, multiple switches and/or routers, and finally will reach its destination. The data packets get properly passed on by each of these network components based on their source and destination, normally represented by the source and destination IP addresses.

As robust as this data transmission process sounds, none of the participating hardware guarantees it will not lose or even purposely drop any packets they handle. It happens quite regularly that all or part of the sent data will get lost somewhere along the way to the destination. This raises the question of how this can be avoided.
2.2.2 Network Protocols

Network protocols describe the rules under which data is being transferred. These protocols include information which is given to the data packets before they leave the sender, and which can be used to reconstruct the original data in case of data loss along the transmission path. If there was never any packet loss, network protocols would be much simpler than they are. Network protocols also include data for quality of service (QoS) control (i.e., giving different jobs different priorities on the basis of their specified importance). But the major effort of network protocols is in packet loss handling.

There are many reasons for packet loss, a fundamental reason being that packet switching networks are shared by multiple users, computers, applications, sessions, etc. Even a single user with a single computer can run many networked programs simultaneously, which will share the network and compete with each other. For example, if a user spontaneously watches multiple videos on Youtube, each sessions might noticeably slow down, while closing some of them will make the remaining ones faster again.

The core routers on the Internet, at any given time, might manage millions of parallel sessions. These sessions compete for resources, and when a packet of one session cannot be scheduled to be delivered properly, it will be dropped. Packet loss happens for many different reasons, e.g., collision in a wireless network environment, switch/router buffer overflow, link errors, bit errors during transfer, receiver buffer overflow, etc. It is commonly believed that the majority of packet loss happens in switches and routers, and that it cannot be eliminated.

Packet loss does not only happen on the public Internet, but also on high-speed dedicated research networks, also known as Lambda Networks. Normally those networks, rather than shared by millions of users, only have hundreds of users at most. Although the number of users is small, the average bandwidth per user is high, so that packet loss still happens and needs to be addressed by communication protocols and the networked software applications.

Although network protocols are mainly designed to cope with packet loss in shared packet-switching networks, the principles of how they work vary, and are largely decided by network settings and application characteristics.
2.2.3 Packet Loss Recovery

When packet loss is detected, the network protocols will react in a pre-defined way. First, the lost data packets need to be recovered, and then further action is taken to reduce the chance of future packet loss, for instance through reducing the data rate. The following is a brief overview of how existing protocols handle packet loss.

1. TCP

The Transfer Control Protocol (TCP) is the protocol used by most Internet applications. Standard TCP starts a transfer at a pre-defined sending rate, and waits for an acknowledgement for each data packet it sends. When acknowledgements are received correctly, TCP will try to increase the sending rate linearly; when no acknowledgement is received within a certain time window for a data packet, it will resend the lost packet and cut the sending rate in half. All packets are identified by their unique, sequential packet numbers assigned to each TCP data packet by the protocol. In principle, TCP guarantees reliable delivery of data packets. TCP manages the sending rate very conservatively, and there exist many variations of TCP [WMJ+04, KHRIT02] which attempt to improve the performance of TCP without losing the fairness of the standard TCP protocol. Nowadays, most network applications or higher level protocols, for instance HTTP on the Internet, are implemented using TCP, which also means that the majority of all network traffic on the Internet uses TCP.

However, TCP is not suitable for high speed data transfer on dedicated high speed fiber networks.

TCP has been used on the Internet for decades. As shown in this paper [Han06], TCP fulfilled the mission of Internet sessions fairly well for our everyday Internet surfing. The speed of TCP is increasing gradually as Internet service providers (ISP) gradually upgrade their switching/routing equipment, so they can gradually increase the bandwidth. In the mid 1990s, it may have taken minutes to load a simple HTML webpage through a 56Kbps modem network, whereas now we can easily view high definition video clips from Youtube with our 10Mbps cable Internet service. However, the performance improvement of Internet applications is
not due to the improvement of the TCP protocol, as the TCP protocol has essentially remained the same since then. Applications benefit from the evolution of network hardware technology. This shows that TCP has so far not been a bottleneck on the Internet, and the performance of TCP keeps improving as computer systems are being upgraded, network link bandwidth is going up, and switching and routing equipment performs better. TCP has so far scaled well on the Internet.

However, when we look at future networks with bandwidths like 1Gbps, 10Gbps or more, TCP is going to break down. Here is an example. On OptIPuter networks, we experimented with a simple file transfer between San Diego and Chicago over a 10Gbps link. Without performance tuning, meaning using the system default settings like send/receive buffer size, data packet size, etc., TCP based file transfer speed between these two cities is below 10Mbps; with performance tuning, by increasing the send/receive buffer size of TCP, enabling jumbo frames (9000 bytes per packet instead of the standard 1400 bytes per packet), the file transfer speed between the two cities can go up to 100Mbps. But still, we only use only a small part of the 10Gbps link. Investigating the reasons for this using bandwidth performance tools (like iperf or nuttcp) shows us that the bottleneck is not the hardware, but it is the transfer protocol (TCP).

TCP will eventually become a bottleneck on the Internet, assuming that the bandwidth of current high speed research networks will be available for home use in a few years. The bandwidth of future networks will exceed the capabilities of TCP. The way TCP handles packet loss with acknowledgements and re-transfer is unsuitable to be used with high speed long distance data transfer. Round Trip Time (RTT) is the time spent for a data packet to travel from its source node to its destination node. RTT characterizes how far two sites are away from each other on the network. RTT is governed by the network link transfer speed and the transmission latency. On typical data networks, the rule of thumb for the speed of data transfer is 200,000km/s. For example, the distance between San Diego and Chicago is around 3000km, so the RTT can be estimated as 3000*2/200,000 = 30ms. In reality, the measured RTT between our machines at Calit2 in San Diego and those at EVL at UIC is about 90ms. The difference comes from the
fact that the actual length of the connecting network between these two sites is longer than a straight line between them; another factor is the latency incurred by switch/router buffering time. Theoretically, two physically co-located machines can have a long network distance if people purposely configure them that way; but two physically distant nodes cannot be close neighbors on the network regardless how they are configured, because of the physical distance between them.

TCP provides a sliding window to solve the transfer speed limits caused by a large RTT. The value of the buffer can be modified to a large value to delay the acceptance of acknowledgements of data packets, which effectively overlap the sending process and the process waiting for acknowledgement, so as to eliminate the idle time of TCP. The optimal value should be set equal to Bandwidth*RTT. In our case, 10Gbps*90ms = 900MB. This means the operating system will have to allocate 900MB of RAM just for using the TCP protocol. And as we showed before, even with tuning, we only get around 200Mbps over a 10Gbps link.

2. UDP-based protocols

Although few applications on the Internet use UDP, on high-speed Lambda Networks many applications use UDP in order to replace TCP. With UDP packets, higher transfer rates can be achieved for long distance data transfer. For example, on the OptIPuter network, we observed that a large portion of the data is transferred via UDP. UDP-based protocols handle packet loss differently than TCP, but no standard has been developed for it yet. The software engineers typically design and implement protocols which support a specific application. We are going to report on a few typical ways of handling packet loss in UDP protocols.

(a) Aggregation Re-transfer Protocols

This type of protocols transfers large amounts of data by first dividing them into large data blocks, in the order of tens or hundreds of megabytes, and the data blocks are then sent out via normal UDP packets. But unlike TCP, rather than waiting for individual acknowledgements for those packets, the sender will wait for a summary from the receiver specifying the list of lost packets in each of the data blocks. The sender then resends the lost packets
as a whole. The successful transfer of a data block may take a few rounds, and once a data block has been correctly delivered, the sender will proceed to the next data block. Compared to TCP, aggregation re-transfer protocols are much faster when transferring very large files. But since they cannot promptly re-transfer the packets at the beginning of each data block, these protocols are seldom used for data streaming. An example for this category is RBUDP designed by EVL at the University of Illinois, Chicago.

(b) Rate-based Transfer Protocols

In a variety of applications like video streaming, data loss below a threshold is unnoticeable with human eyes. Taking advantage of this property, some protocols focus on controlling the data loss rate by using an adequate data sending rate, and once they find signs of packet loss, they will reduce the sending rate. The key to the success of this category of protocols is the prediction or early detection of potential packet loss, because it will be too late to wait for the receiver to report data loss, particularly in long distance high-speed data streaming. For instance, LambdaStream predicts packet loss by checking the receiving time stamp of consecutive packets. If the time gap between two consecutive packets increases at the receiver, it will predict that the queuing time in the switches increases, indicating the packet buffers in the switches are filling up. If this trend continues, the switch buffers will eventually be full, and packets sent to a full buffer will be dropped. To avoid this, LambdaStream reduces the sending rate at the sender until it detects the time gaps of consecutive packets received by the receiver start to decrease, indicating the queue buffers in the switches are getting shorter. LambdaStream cannot guarantee reliable delivery of data, but it can very well control the data loss rate under a small threshold, which makes it feasible for point-to-point multimedia data streaming.

(c) Error Correction Methods

Packet loss does not always lead to unreliable transmissions. Error Correction is a technique using redundant information to recover partial data missing. RAID is a well-adopted error correction method to recover disk
2.2.4 Forward Error Correction

Forward Error Correction (FEC) is “a mechanism of error control for data transmission, whereby the sender adds redundant data to its messages, also known as an error correction code. This allows the receiver to detect and correct errors (within some bound) without the need to ask the sender for additional data.” [ANAS]. Since most high bit-rate transfer protocols use unreliable UDP packets to deliver data, they often use a certain type of FEC to recover lost data. FEC can correct a small percentage of random packet loss, depending on the amount of FEC data sent, but is NOT always effective for burst packet loss over a certain bound.

In recent years, a new series of FEC coding techniques have been invented, called Fountain Code [Fou], which “can generate a limitless sequence of encoding symbols from a given set of original symbols such that the original symbols can be recovered from any subset of the encoding symbols of size equal or only slightly larger than the number of original symbols.”. In the network communication context, a symbol usually represents a UDP packet.

All FEC codes come with an overhead. The overhead of the fountain code is called “erasure rate”, which is defined as \( \varepsilon \), meaning if \( N \) is the number of original symbols, \((1+\varepsilon) \cdot N \) encoding symbols will be sufficient to recover the \( N \) original symbols. Fountain code only works for large value \( N \), so it is impossible to give out a simple fountain code example. But in principle, the encoding/decoding processes of fountain code are based on a bipartite graph, in which the edges are generated by a degree distribution function [Lub02]. The three forms of representation of fountain codes are shown in Figures 2.1: the formula, the bipartite graph and the matrix. These forms are equivalent. In this thesis, we use the matrix representation form because it best illustrates the encoding and decoding processes. Examples of fountain code include online code, LTCode, and
(a) $V_1$-$V_4$ are original data packets; $Y_1$-$Y_5$ are encoded data packets. The formula and bipartite graph have the same meanings.

(b) Matrix element $M_{i,j}$ is marked to black iff data packet $V_i$ is included in code packet $Y_j$.

**Figure 2.1**: Representation forms of fountain code

Raptor Code [Sho06].

Compared to other fountain code techniques, Raptor Codes are the first known classes of fountain codes with linear encoding and decoding time. Senders and receivers
must use the same data structures to encode or decode the data, including the same pseudo-random number generator, and the same coding matrices.

2.3 Will Parallel Data Transfer Be The Trend?

As CPUs start to have more and more cores instead of being faster and faster, as GPUs start to integrate hundreds of processing units, we have seen a clear trend from the industry that parallelism is the right direction to break the limit of the processing capability of a single unit.

So, from previous analysis, we also envision a parallel data transfer model in some future network applications. We do not know whether parallel data transfer will for sure be the trend, but we explore the possibility of using parallel data streams to fit the requirements of future network applications in the first place.

In the following Chapter, we will explain why parallel data transfer is a hard problem, and discuss how FEC codes are helpful but not sufficient for large-scale parallel data streaming protocols. We then introduce the new Cross-Stream coding approach to perform parallel data transfer.
Chapter 3

Parallel Data Transfer Using Cross-Stream Coding

3.1 Introduction

Transferring very high quality digital objects over optical networks is critical in many scientific applications, such as video streaming/conferencing, remote rendering on tiled display walls, or 3D virtual reality. Current data transfer protocols rely on UDP as well as a variety of compression techniques. None of the existing transfer protocols, however, scale well to many parallel data connections. Existing parallel streaming protocols have limited synchronization mechanisms for multiple streams, and they are prone to be slowed down significantly if one stream experiences significant packet loss. In this paper, we propose a new parallel streaming protocol which can stream many parallel data streams over optical networks: CSTP, the Cross-Stream Transfer Protocol. It not only tolerates random UDP packet loss, but also aims to tolerate unevenly distributed packet loss patterns across multiple streams to achieve synchronized parallel streams with limited coding overhead. We simulated the approach, and the results show that CSTP can generate steady throughput with fluctuating data streams of different data loss patterns, and can transfer data in parallel at a higher speed than multiple independent UDP streams.

In summary, point-to-point or point-to-multipoint protocols are not sufficient
to feed increasingly larger displaying devices, and multipoint-to-multipoint protocols are difficult to efficiently synchronize. In this Chapter, we examine the multiple-to-multiple communication model, and present the new Cross-Stream(CS) coding mechanism, which focuses on the synchronization challenges of the parallel data storage/retrieval/communication model. We ran experiments with our new protocol, and the results show that even with fluctuating network throughput, or unevenly distributed packet loss in parallel streams, our approach can achieve reliable synchronized throughput. In Section 3.2 we discuss the challenges of the parallel data usage model; in Sections 3.3 and 3.4, we discuss our approach for reliable synchronized parallel streaming. We show simulation results in Section 3.5. In Section 3.6, we conclude and suggest future research directions.

In the following section, we will explain why synchronized parallel data transfer is a hard problem, and discuss how FEC codes are helpful but not sufficient for large-scale parallel data streaming protocols.

### 3.2 Parallel Data Transfer Challenges

As an example, 4K video streaming is very bandwidth intensive and a good motivating application for our parallel streaming algorithm. 4K was defined by the Digital Cinema Initiative (DCI) consortium of Hollywood studios in 2003. It describes video with up to 4096 pixels per line and up to 2160 lines per frame. In many practical applications, and in the remainder of this thesis paper, 4K refers to a frame size of 3840x2160 pixels, which is exactly four times the resolution of 1080P HDTV. 4K video can be captured by 4K cameras, which store every frame as an image file, for instance TIFF. And 4K projectors can display this material natively. Streaming uncompressed 4K video is challenging because the data bit-rate is very high, namely, the bit-rate for 32bit per pixel 4K images at 24fps is over 6Gbps, which does not fit into one Gigabit channel, but could be sent through multiple parallel Gigabit connections. So alternatively, assuming we have a large number of 4K frames to send, we can use multiple senders to stream these images to multiple remote receivers. Each of the senders sends out a portion of the images in UDP packets, and each of the receivers receives the corresponding parts
of the images and displays these parts on part of a tiled display wall. And the technical challenges become more severe when the scale of the problem increases.

In an ideal situation, if all parallel streams are reliably delivered, there will be no problem with this approach. But in reality, UDP data packet loss is inevitable in such high-bandwidth applications, which will lead to missing data and cause synchronization problems at the destination. To our best knowledge, there are no existing through research about the UDP packet loss pattern of large scale parallel data streams over optical networks; but based on our experience and experiments over the OptIPuter network, UDP packet loss behavior is nondeterministic, and is a function of many variables, such as the destination nodes hardware configuration, their workload, OS type and version, OS scheduler, NIC configurations, switch/router scheduling, switch/router queuing strategy, switch/router buffer size, and many more. In particular, if there are multiple streams on the same fiber link nearly saturating the capacity of the link, we found the following typical UDP packet loss behavior over the link:

1. Very frequently, burst packet loss for an individual stream may happen and will not be recovered by simple FEC code. We can either ignore that which will lead to missing data, or we can go back to the sender to request resending the lost packets, which is time consuming and will make other streams wait.

2. Bandwidth is not evenly shared by the streams for many reasons. For instance, the UDP protocol does not have constant execution time, neither does switch/router scheduling or queues in switches/routers; the end nodes’ capability may vary with time; occasional interference from other network applications may make data streams fluctuate, etc.

3. The overall/average packet loss rate of all data streams is stable, which means the switching capabilities are stable.

Figure 3.1 shows experiments verifying packet loss behavior and packet loss distribution of parallels streams over CaveWave, a 10Gbps dedicated optical network between Chicago and San Diego. In Figure 3.1a, the result of fourteen parallel “iperf” tests on an oversaturated 10Gbps connection is shown. As expected, an average packet
(a) Parallel iperf experiments between Chicago and San Diego.  
14 streams, 750Mbps/stream (10.5Gbps) over 10Gbps optical network.  
Average packet loss rate is around 6% in all experiments.  
X-axis: five independent experiments.

Figure 3.1: Parallel data loss pattern
loss rate of 6% is observed in each of the five independent experiments, but packet loss rate patterns and distribution vary significantly between experiments. Figure 3.1b is a plot of a similar experiment using video streaming software to measure the frame-by-frame packet loss rate and to understand the burst packet loss pattern. It shows that burst packet loss occurred frequently and in arbitrary streams.

Our experiments confirm that using multiple data streams to transfer data between multiple senders/receivers brings along new challenges. FEC code is helpful in many cases, but not sufficient to handle burst packet loss for individual streams. In video streaming, losing a few packets might be acceptable and we can use interpolation methods to simulate the lost pixels and probably will not affect the image quality a lot, as long as the packet loss rate is low. But in other scientific applications, for example, earthquake simulation and visualization, lost packets may contain critical data that cannot be simulated.

![Diagram of water pouring](image)

(a) Four faucet are pouring water into four containers. Because the water streams differ, containers, and the water levels are even despite the water streams remain.

(b) Now, we install pipes between adjacent containers. Because the water levels in the containers differ.

**Figure 3.2**: An analogical example of water pouring

Considering a model similar to the water-streaming model, as shown in Figure 3.2. In Figure 3.2a, four faucets are pouring water into four containers. Because water can splash out of the containers, the water levels are not even in four containers. Fortunately, we have a simple solution for this problem as shown in Figure 3.2b, pipes are added between adjacent containers, so that the water levels are automatically equalized. Although this model is not directly applicable to our parallel data streaming model, it provides an important insight that we might be able to tightly couple data streams to design a correlated parallel data streaming protocol instead of a protocol consisting of
multiple independent data streams.

In the next section, we will discuss our new Cross-Stream Coding mechanism and the transfer protocol using Cross-Process Coding.

### 3.3 Cross-Stream Coding

Loosely coupled streams are hard to coordinate; linking data streams together allows us to consider new methods to correct for single stream burst packet loss. If one data stream experiences burst packet loss, the receiver does not have to fetch the lost packets from its sender, instead, it gets the data from its peers at the receiving side. Taking into account the Round Trip Time (RTT) to fetch a bit from remote senders, being able to get it locally is a significant advantage. However, it is difficult to couple parallel data streams because they are essentially independent data streams carrying uncorrelated data. To achieve our goal, we first must correlate the data in the data streams. By applying appropriate coding, the encoding strategy of our parallel streaming protocol needs to be able to:

1. tolerate a certain percentage of packet loss
2. encode and decode efficiently
3. encode multiple streams to carry information between them

Thus, we selected a fountain code as the basis of our work. Fountain code is a type of FEC code which is often used in wireless communication environments where receivers are sometimes not capable of sending back acknowledgements to request the resending of lost packets. In computer science, fountain codes have been used for high-bandwidth multicast [CDK+03], the robust storage system RobuSTORE [XC07], etc. But to our knowledge, it has not been used in high bandwidth parallel data transfer applications. In the remainder of this Chapter, we will explore and analyze the coupling of parallel streams using fountain code, introduce our new Cross-Stream(CS) coding scheme, and we will discuss the implementation of a transfer protocol using CS coding in detail.
The input of the scheme is a sequence of original data frames, for example, a series of video frames, and the goal is to transfer these data frames from multiple senders to multiple receivers synchronously. Each original data frame is composed of $S$ equally sized sub-frames that will be sent out by $S$ individual senders to their corresponding receivers. This subdivision does not always mean additional computational cost, particularly when the sub-frames are acquired independently, for example, by $S$ independent cameras. Each sub-frame is further decomposed into $N$ equally sized original data packets, and the original data packets are encoded into regular code packets using fountain code. In addition to the regular code packets, we encode a certain percentage of special code packets, which are XOR versions of regular code packets from different sub-frames, and we call them CS code packets. The senders send out the combination of both regular code packets and CS code packets, and upon receiving those code packets, the receivers use the regular code packets to decode the original data and use CS code packets to communicate between streams to help them reconstruct missing packets.

### 3.3.1 Simple Cross-Stream Coding Mechanisms

We first briefly explain CS coding using two straightforward ways.

1. Unified Parallel Code

In the video streaming example, each image can be divided into equally sized $N$ original data packets. The unified approach generate $(1 + \beta) \cdot N$ coded packets from $N$ original data packets, and evenly distributes them to four receivers, and as long as the sum of received packets of all four receivers exceeds $(1 + \varepsilon) \cdot N$ ($\varepsilon$ is the overhead of the fountain code, $\varepsilon < \beta$), the image can be recovered.

The issue of this approach is the very high coding communication cost, both for encoding and decoding. To be specific, in the decoding process, where once an original data packet is decoded, it has to be distributed to all receivers that have received code packets containing that data packet. Since the possibility of one data packet to be included in multiple code packets is very high in Fountain Code, this overhead will significantly slow down the decoding process. Moreover, once the decoding process is completed, the data needs to be relocated to the right
receivers. As an example, when we show the 4K images on four HD displays driven by four receivers, we need R1 (Receiver 1) to have the upper-left part of the image, R2 to have the upper-right part of the image, and so on. But, during the decoding process, although the decoders have recovered all the original data packets, those data packets are randomly distributed across the receivers and need to be re-distributed to the right receiver based on their positions in the image. The decoding communication cost is on the order of the total number of original data packets.

2. Cross-Distributed Code

Alternatively, we can first decompose each 4K image into sub-images, and separately encode them into encoded sub-images. When this is done, rather than sending each encoded sub-image to one receiver, we can first mix the encoded sub-images together, and send a portion of the mixed data to each receiver. Once the receivers have received their portion of the mixed data, they first communicate to reassemble their corresponding encoded sub-images and then start to decode independently.

The benefit of this approach is that even if one stream lost many packets, those lost packets are likely evenly distributed among all encoded sub-images, so that all receivers can successfully decode their encoded sub-images with high possibility. But the problem is this approach still introduces very high communication cost on both sides. Even when no packet loss, the cost of this approach is still very high.

The two straightforward approaches to couple parallel data streams have fundamental problems and cannot be practically used in real applications. So we get back to the water stream model and couple parallel data streams in a similar way using pipes. The original data are encoded in two dimensions: one dimension is along local original data packets; the other dimension is along parallel data streams. That is, each CS code may have two-dimension degrees: a stream degree and a few fountain code degrees. The construction of CS code is the basis of our Cross-Stream Transfer Protocol(CSTP).
Table 3.1: Definitions in CSTP

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>A data communication between one sender and one receiver</td>
</tr>
<tr>
<td>Data Packet</td>
<td>A data block of a sub-frame in each of the stream</td>
</tr>
<tr>
<td>Regular Code Packet</td>
<td>A fountain code block of a sub-frame in each of the stream</td>
</tr>
<tr>
<td>CS Code Packet</td>
<td>A Cross-Stream fountain code block of a sub-frame in each of the stream</td>
</tr>
<tr>
<td>S</td>
<td>Number of Parallel Streams</td>
</tr>
<tr>
<td>N</td>
<td>Number of original data packets of a sub-frame in each of the stream</td>
</tr>
<tr>
<td>ε</td>
<td>Erasure rate of fountain code</td>
</tr>
<tr>
<td>α</td>
<td>Targeting tolerance rate of code packet</td>
</tr>
<tr>
<td>$N'$</td>
<td>Number of regular code packets of a sub-frame in each of the stream, to be able to tolerate $\alpha$ loss, $N' = \frac{N}{1 - \alpha}$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$\frac{N'}{N} - 1$: Overhead of regular fountain code</td>
</tr>
<tr>
<td>$N''$</td>
<td>Number of all code packets of a sum-frame in each of the stream, including all regular code packets and CS code packets</td>
</tr>
<tr>
<td>δ</td>
<td>Overhead of CS Coding, $\delta = \frac{N''}{N'} - 1$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Overhead of all code, $\gamma = \frac{N''}{N} - 1$</td>
</tr>
<tr>
<td>$M_{1...S}$</td>
<td>Fountain code generating matrices of S data streams which are known to all senders and receivers beforehand</td>
</tr>
<tr>
<td>$V_{[1...S][1...N]}$</td>
<td>The original data packets in S streams</td>
</tr>
<tr>
<td>$Y_{[1...S][1...N'']}$</td>
<td>The regular/CS code packet in S streams</td>
</tr>
<tr>
<td>$d_F$</td>
<td>Code Degree: the number of XORed data packets in one regular code packet</td>
</tr>
<tr>
<td>$D_F$</td>
<td>Code Degree Distribution: the degree distribution function of the regular fountain code</td>
</tr>
<tr>
<td>$d_S$</td>
<td>Stream Degree: equals 1 for a regular code packet, or 2...S for a CS code packet, meaning how many streamed a CS code packet connects</td>
</tr>
<tr>
<td>$D_S$</td>
<td>Stream Degree Distribution: the distribution of $d_S$. For example, when S is small, we use $D_S(x) = \frac{\delta}{1+\delta} \cdot x^S + \frac{1}{1+\delta} \cdot x$. This function encodes all CS code packets with the same stream Degree S.</td>
</tr>
</tbody>
</table>
3.3.2 CSTP: Cross-Stream Transfer Protocol Via CS Code Packets

1. Definitions

The definitions used in our protocol are summarized in Table 3.1.

2. Generating Matrices: $M_{[1...S]}$

$M_{[1...S]}$ are pre-generated generating matrices for CSTP. Similar to fountain code generating matrices shown in Figure 2.1a and Figure 2.1b, $M_i$ is basically a standard fountain code matrix with extra CS code packet columns. So, to generate $M_i$, we first create a standard fountain code matrix, then insert CS code packet columns into the matrix. Each CS column is an assembled column of multiple columns from the $S$ standard fountain code matrices of $S$ streams. A sample is shown in Figure 3.3. A CS code packet can consist of regular code packets from all streams, or only two streams. In the $D_S$ function in Table 3.1, all CS code packets consist of regular code packets from all $S$ streams. We will use a different $D_S$ in our experiments in Section 3.5, with which distribution function, 50% of the CS code packets have a stream degree of $S$, and 50% have stream degree 2. A CS code packet with a stream degree of 2 will consist of one regular code packet of its own and one regular code packet randomly selected from the other $(S-1)$ data streams.

3. Encoding and Decoding Algorithms

(a) Encoding Algorithm

For each encoder $[1...S]$:

Step 1: Break its original sub-frame into equally sized data packets.

Step 2: Get next column from $M_i$, if it is a regular code packet, XOR all packets that are valid in that column; if it is a pipe packet, fetch data from other encoders and XOR all packets that are valid in that column.

Step 3: If enough packets were generated, stop, else go to Step 2.

(b) Decoding Algorithm
Figure 3.3: Sample matrices. The portion in the big circle marked on M2 identified a regular fountain code matrix; the chained four small circles identified a pipe packet in M3.

S=4, N=4, N’=5, N”=6

X-axis: code packets Y[4][6]
Y-axis: original data packets V[4][4]
CS code packets (marked in red): Y₁₃, Y₂₅, Y₃₂, Y₄₄

---

For each decoder [1...S]
Step 1: Run a native fountain code decoding algorithm.
Step 2: Use pipe packet code to communicate with other decoders: if data needed, request data from other decoders, if data available send to other decoders.
Step 3: Repeat Step 2 until all data are decoded or the decoding process halts.

Figure 3.4 illustrates how the decoding process works, and how CS code packets can help receiving nodes recover lost packets. Assuming receivers R1, R2 and R4 have received enough packets, and R3 has lost a big portion of the packets, R1, R2 and R4 can recover their data by applying the standard fountain code decoding algorithm, as illustrated in Figure 3.4a. When they finish the decoding process, all the code packets in the their CS code packets have also been decoded through
(a) Assuming R1, R2 and R4 have received all the packets and successfully recovered their original data (marked as green), but R3 lost two code packets: $Y_{34}$ and $Y_{35}$ and need extra to decode.

(b) The decoded data packets are distributed to each other via CS code packets, and all CS code columns now become regular columns of M3.

(c) R3 collects CS packets from R1, R2 and R4, and now can decode its original data.

**Figure 3.4:** Decoding procedure
local communication, shown in Figure 3.4b, and the remaining parts of these pipe packets are extra code packets for R3. These extra code packets are sent to R3 so that R3 now will have enough code packets for decoding, as shown in Figure 3.4c.

3.4 CSTP Implementation

Data transfer involves two sides: senders and receivers. In a classical client-server communication paradigm, server open network ports to accept incoming connections, and clients connect to the pre-defined server addresses and ports. The connections are then established and communication start. Both servers and clients are capable of sending and receiving data according to the application’s communication model.

In CSTP, the paradigm is the same. We define senders and receivers in the same way as a typical client-server program will do: the servers open UDP ports, and the clients connect to the UDP ports. The servers continues sending out blocks of data to the clients, and the clients will receive them and handle them. Blocks of data can be part of big files, or series of files. As in CineGrid, the data we have are lots of image files, we use this case in our experiments. Not to lose generalization, all files or data blocks are with a same pre-defined size, which is a adjustable parameter.

Without regard to CSTP, the streaming part of our program is one to one communication model. Which means one client only connect to one server, and one server only send data to one client. The servers are not associated with each other, neither the clients. As we will discuss later, all the connections among servers or clients are part of the encoding/decoding modules in our CSTP protocol.

Therefore, the data transfer part is a single source single destination network data transfer protocol. As shown in figure. The client side is no more complex than reading and sending out blocks of data at a pre-defined rate; the server side will use a ring-buffer to buffer the data locally and pass the data to whatever modules that are above it in the protocol.

CSTP uses multiple senders and multiple receivers. The goal is to send sequential data frames from the senders to the receivers synchronously. Each data frame is initially divided into multiple sub-frames, and each sender sends out just one sub-frame
to its receiver. The senders keep reading sequential data sub-frames, encode them and send the encoded sub-frames to their corresponding receivers; the receivers receive the data perform Cross-Stream decoding to reconstruct the original data. The senders and receivers communicate via a high-speed network.

### 3.4.1 CSTP Senders

The sending procedure consists of three major functions: reading, encoding and sending, which are controlled by a timer-based synchronizer. The senders are not the bottleneck in the protocol.

![Code matrix data structure](image)

**Figure 3.5:** Code matrix data structure

The CSTP encoding process is implemented using a coding matrix, as shown in Figure 3.5. The coding matrix is a two dimensional (2D) linked list, associated with four memory buffers: DataBuf, CodeBuf, PipeBuf and ExtraBuf. Each node \((i, j)\) in the 2D
linked list means code packet \( i \) contains data packet \( j \) in its XOR list. DataBuf stores the original data sub-frame read from the input devices, e.g., a partition of the video frame file, or input from one of multiple cameras. CodeBuf and ExtraBuf are generated from DataBuf using the coding matrix (also see Figure 2.1). CodeBuf includes regular code packets and pipe packets; ExtraBuf includes only regular code packets for the sending nodes to construct their pipe packets. PipeBuf are regular code packets received from the senders. Each packet in ExtraBuf will be sent to the PipeBuf of one of the other senders.

A regular code packet (only composed of local data packets) in CodeBuf is encoded immediately from DataBuf. A pipe packet that contains packets in PipeBuf will wait until that packet is received from the peers. Practically, the senders will encode ExtraBuf first and distribute the ExtraBuf packets to its peer senders so that the encoding of CodeBuf will not need to wait for the PipeBuf packets. Therefore, the encoding process is a deterministic process. Finally, the sender only sends out the code packets in CodeBuf to the receiver, while ExtraBuf is not sent.

### 3.4.2 CSTP Receivers

The receivers have two major threads: a streaming thread and a decoding thread. They operate on two major data structures: a circular buffer and a coding matrix.

1. **Circular Buffer**

   A circular buffer is used to buffer the received data sub-frames. It has two pointers: a reader’s pointer and a writer’s pointer, controlled by the reading and writing thread respectively. The writing thread (the stream thread) reads UDP packets from the network and saves them into the blocks pointed to by the current writer’s pointer. The reading thread (the decoding thread) fetches a block from the circular buffer and sends it to the decoder for decoding. Both threads will increase their circular buffer pointers after finishing writing/reading.

2. **Decoding Matrix** Although the decoding matrix data structure is the same as the encoding matrix data structure, the decoding algorithm is much more involved. After the reading thread fetches a sub-frame from the circular buffer, it will fill
the decoding matrix’s CodeBuf and associate with it a Mask Variable, which is an array of “Boolean” values, marking each packet in the CodeBuf either “Lost” or “Received” so that only “Received” packets will be used for decoding. Because the packet loss is not known beforehand, the decoding process is non-deterministic.

In the decoding process of standard LT Code or Raptor code, the decoder keeps looking for packets in CodeBuf whose remaining code degree equals one, decoding that packet by copying the code packet to the data packet, and distributing the decoded data packet to all other code packets containing that data packet, until the process stops without any further possible movement.

In our algorithm, we first perform this standard decoding process based on CodeBuf and the Mask Variable using the decoding matrix. After this is done, a decoder either has recovered all its packets in DataBuf, or only partially decodes DataBuf due to the packet loss in CodeBuf. If all decoders have successfully decoded their sub-frames, the whole data frame is successfully received and the algorithm can proceed to the next data frame.

If some receivers cannot recover all their data packets, the decoders will need to work together to recover the un-decoded parts. ExtraBuf and PipeBuf are used to communicate between decoders to decode those parts through pipe packets.

(a) From PipeBuf to ExtraBuf:

Assuming receiver A has received a pipe packet \( Y = V_{A,1} \oplus V_{A,2} \oplus \ldots \oplus V_{A,K} \oplus Y_B \), and has decoded all its data packets \( V_{A,[1..k]} \), it can now decode \( Y_B \) by doing \( k \) times XOR operations on \( Y \), and \( Y_B = Y \oplus V_{A,1} \oplus V_{A,2} \oplus \ldots \oplus V_{A,K} \). \( Y_B \) becomes a new regular code packet in A’s PipeBuf for receiver B, so that it will be sent to B’s ExtraBuf to help B decode un-decoded data packets as if B receives one more regular code packet from its sender.

(b) From ExtraBuf to PipeBuf:

In each decoder, ExtraBuf is set to “zero” before the decoding process. During the first phase of the decoding process, in addition to the standard decoding process, all the decoded data packets are also distributed to the associated
code packets in ExtraBuf, and the code packets in ExtraBuf will “XOR” the
data packet to itself. Therefore, for those decoders which have successfully
decoded all their data packets, they also have generated the code packets in
ExtraBuf. As explained before, ExtraBuf are used by the encoder to encode
a pipe packet during the encoding process and are not sent to the receivers.
Just as the encoder sends the code packet in ExtraBuf to its peers to generate
a pipe packet in the peers, now the newly generated code packets in Extra-
Buf are sent to the same pipe packet. For example, assuming a pipe packet
Y is located in receiver A, and \( Y = V_{A,1} \oplus V_{A,2} \oplus \ldots \oplus V_{A,K} \oplus Y_B \oplus Y_C \oplus Y_D, \)
\( Y' \) becomes a new code packet for A, as if A had received a new code packet
from its sender.

ExtraBuf and PipeBuf are mutually beneficial. In practice, Step a) and Step
b) will start as early as possible, overlapping with the self-decoding process,
and will repeat in the post decoding process.

### 3.4.3 Reliable Delivery

In our CSTP implementation, the goal is to tolerate unevenly distributed burst
packet loss below a certain threshold, but when significant packet loss happens for many
streams, CSTP cannot guarantee the reliable delivery of the data. Based on our experi-
ments, CSTP tolerates burst packet loss quite well and is sufficient for reliable delivery
in many practical scenarios. However, in case packet loss is significant for many data
streams, and after the cross-stream decoding process there are still un-recovered data
packets left, more data would have to be requested from the senders if reliable deliv-
ery is required. Two categories of issues have to be discussed: dynamic sending rate
and lost packets resending. We have not covered these in our CSTP implementation,
but they could be integrated in future versions of the algorithm. Dynamic sending rates
have been studied in RBUDP, UDT and LambdaStream, and we have a similar issue
in CSTP. In order to resend lost packets, other approaches maintain a bitmap for lost
packets and re-transfer lost packets, which is straightforward but ineffective, because the senders have to keep all data in main memory to avoid random disk access. For big file transfers, RAM will be used to keep the original data quickly accessible, like in RBUDP. In CSTP, instead, we only need to keep a small percentage of extra code packets in the RAM of the senders. So when receivers ask for more data, the senders simply send out these new code packets. This is less expensive than saving all original data for re-transfer.

3.4.4 Scalability of CSTP

The scalability of CSTP is primarily subject to the communication cost of CSTP. In CSTP, both encoding and decoding algorithms will incur certain communication cost among the nodes of a cluster. In the encoding process, the communication is deterministic, while decoding communication is more nondeterministic because the decoding process is subject to packet loss. But on both sides, the communication cost of the CSTP is reasonably small when $S$ is small because only pipe packets are needed to communicate among the nodes. In the worst case, the total communication cost has an upper bound of $p$, e.g., if $p=0.05$, $S=4$, the decoding communication cost is about 15% of the size of the original data.

The scalability will be more important when $S$ is bigger. For small numbers of parallel streams, the stream degree distribution function in Table 1 works quite well. However, for larger scale applications, which use more than ten parallel streams, the communication cost with that distribution function will increase linearly with the number of streams. Therefore, we are designing and experimenting with different distribution functions. Our preliminary results show that with constant communication overhead, the CSTP protocol scales very well to larger numbers of parallel machines, with reasonable assumption of parallel data loss pattern.

Another factor that can affect the scalability of CSTP is the synchronization of large numbers of nodes. In applications requiring frequent synchronization operations, more thought needs to be put into this. In our current experiments, the synchronizer in CSTP can complete more than 600 synchronizations per second among 15 machines. With 100 machines, that number is about 360 per second. Therefore, we do not foresee
this as a limiting factor.

3.5 Experimental Results

In our experiments, we evaluated the decoding performance of the most recent version of the fountain code, and simulated multiple data streams using cross-stream coding algorithms, test the stability of decoding algorithms with fluctuating data flows and various packet loss rates, and compare the transfer speed of CSTP with parallel RBUDP. The fountain code we selected is the Raptor Code, which has linear time encoding and decoding algorithms. The pre-code and LTCode we used are those described in [Rap]:

Precode: LDGM+HDPC LTCode degree distribution:

\[ D_F(x) = 0.0156 \cdot x^{40} + 0.0797 \cdot x^{11} + 0.111 \cdot x^{10} + 0.113 \cdot x^{4} + 0.210 \cdot x^{3} + 0.456 \cdot x^{2} + 0.00971 \cdot x \]

The average code degree of LT-Code is 4.6116, and the average degree of Raptor Code is the sum of LTCode degree and pre-code degree, which is a function of N, e.g., when N=1024, Raptor Code has an average code degree of 13. The average code degree determines the complexity of the decoding algorithms. Theoretically, assuming the packet size is fixed, the complexity of the decoding algorithm is \( O(D \cdot N) \), where D is the average degree of Raptor Code, and N is the number of original data packets.

The platform we used for our experiments is a 2006 Dell XPS 720, with:

- CPU: Intel(R) Core(TM)2 Quad CPU @ 2.40GHz
- Cache size: 4096KB
- Memory: 4GB
- OS: 64bit Version of CentOS 5

3.5.1 Raptor Code Decoding Performance

An important measurement is with how much overhead Raptor Code can recover the original data. Although this has been discussed in prior work, we have done our own experiments as shown in Figure 3.6. In our tests, we find out that N cannot be too small since the Raptor Code is based on probability theory and will not hold for small values
Table 3.2: Decoding speed of the Raptor Code

<table>
<thead>
<tr>
<th>number of Original Packets</th>
<th>number of Raptor Code Packets</th>
<th>Decoding Percentage</th>
<th>Decoding Speed (7680B/pkt)</th>
<th>Decoding Speed (3840B/pkt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>650</td>
<td>100%</td>
<td>2.51Gbps</td>
<td>3.17Gbps</td>
</tr>
<tr>
<td>1024</td>
<td>1200</td>
<td>100%</td>
<td>2.00Gbps</td>
<td>2.22Gbps</td>
</tr>
<tr>
<td>2048</td>
<td>2400</td>
<td>100%</td>
<td>1.68Gbps</td>
<td>1.82Gbps</td>
</tr>
<tr>
<td>4096</td>
<td>4800</td>
<td>100%</td>
<td>1.45Gbps</td>
<td>1.52Gbps</td>
</tr>
</tbody>
</table>

of N. When N is around 1000, Raptor Code can recover original data with an overhead of around 6-8%.

![Raptor Code Decoding Results](image)

**Figure 3.6**: Raptor code overhead

To test the decoding speed of the Raptor Code, we run our implementation of it (there is no open source implementation of the Raptor Code) with one thread to decode native Raptor Code, and the decoding speed is shown in Table 3.2. We tested 7680 and 3840 bytes packets.

The decoding algorithm is CPU intensive, and consumes significant memory bandwidth. Currently in our single thread experiments, the decoding algorithm is capable of decoding a single HD stream. The results also suggest that the cache miss rate is an important factor. As shown, when we increased the number of original packets, the decoding speed went down. We believe this is because the cache miss rate became higher with the larger data set.
3.5.2 CSTP Recovery Performance

First, we need to understand whether streams can help each other when one stream loses significantly more packets than others. We use the following settings:

\[\delta = 5\%\]
\[S = 4\]
\[N = 1024\]
\[D_S(x) = \frac{\delta}{1 + \delta} \cdot x^S + \frac{1}{1 + \delta} \cdot x\]

We verify the performance when single stream packet loss is significant by sending data to four receivers and purposely drop 20% packets from one stream, and test how the other streams can help it recover lost packets to maintain synchronized throughput and measure the protocol overhead of the 2D Cross-Stream Code. Figure 3.7 shows the lossy stream can be recovered with pipe packets from other streams.

![Figure 3.7: Protocol overhead can be calculated from this figure. The original number of data packet is 4340 including pre-code, Raptor Code overhead can be read from Figure 3.6, which is about 7.5% for N=1024, and from this figure, the overall recovery percentage becomes 1 when \(N''=1230\), so protocol overhead is \(\frac{1230 \times 4}{4340 \times 1.075} - 1 = 5.45\%\).](image)

In our next experiment, we tested how CSTP works with random packet loss across all streams. We encoded 1000 frames into four sets of code packets using four generating matrices, and then simulated sending a certain number \(N'\) of code packets to
each receiver. We did not assume all the packets were correctly received, but purposely dropped some of them. And based on our network assumption, the packet loss rate of each receiver is randomly generated with every new frame. The average packet loss rate of the receivers is set to 2%, 3% and 4% respectively.

We then plotted a figure with the X-axis representing $N'$, and the Y-axis representing the possibility of successful recovery of the full frame. The result is shown in Figure 3.8. It shows that with a reasonable overhead, e.g., 5% to 15%, our approach can recover significant packet loss in all streams and the possibility of fully recovered frames is 99.99%.

**Figure 3.8:** These lines show the function of frame recovery percentage to the number of issued packet to receivers, and the lower figure is the zoom-in of the top-right part of the upper figure. (X-axis: Number of code packets; Y-axis: Percentage of recovered original packets)
3.5.3 Burst Packet Loss Recovery

By carefully controlling the sending rate, we are able to deliver data with only a small average packet loss percentage. Ideally, if all packet loss were completely random and evenly distributed across all streams, simple FEC would solve the problem. But in practice as shown in Figure 3, packet loss patterns are not uniformly distributed, and FEC cannot recover burst packet loss. Here we compare the recovery capability of native FEC code and CSTP under a variety of assumed packet loss patterns. We use 10 parallel data streams to do the comparison with an assumption of an average packet loss (APL) of 7.5%. We chose this APL to make the comparison more interesting. The settings of CSTP are:

\[
\delta = 5%
\]
\[
S = 10
\]
\[
N = 1024
\]
\[
D_S(x) = \frac{\delta}{1 + \delta} \cdot (0.5 \cdot x^S + 0.5 \cdot x^2) + \frac{1}{1 + \delta} \cdot x
\]

The meaning of \(D_S(x)\) is: for the pipe packets, 50% have stream degree S, 50% have stream degree 2.

The four packet loss patterns we experimented with are:

1. Unified Pattern: all packet loss is evenly distributed across all data streams.
2. Linear Pattern: packet loss of all data streams fits into a linear pattern.
3. Polynomial Pattern: packet loss of all data streams forms a polynomial pattern.
4. Random Chop: we treat the overall packet loss like a pie, and randomly chop the pie to divide it into 10 parts.

Our experiments show a significant advantage of CSTP over native FEC code. As shown in Figure 3.9, in all four cases, CSTP can recover 100% of the original data in all 10 data streams. But the independent FEC code fails in some data streams in all cases. With packets loss evenly distributed, the FEC code only recovers about 90% of
the original data in two streams. With more unevenly distributed burst packet loss the FEC code’s performance is much worse than CSTP.

Figure 3.9: CSTP vs. pure FEC (X-axis: data stream ID from 1 to 10; Y-axis: packet recovery percentage)

### 3.5.4 CSTP Parallel Transfer Performance

We then compared the transfer speed of CSTP to the popular UDP transfer protocol RBUDP. To measure the protocol performance and avoid disk operations, we modified the RBUDP protocol by eliminating the disk I/O operations, and compared both protocols using 2-10 parallel streams between EVL in Chicago and Calit2 in San Diego.
The link between them was a 10Gbps optical network. The settings of CSTP are:

\[
\delta = 5\%
\]

\[
S = 2...14
\]

\[
D_S(x) = \frac{\delta}{1+\delta} \cdot (0.5 \cdot x^2 + 0.5 \cdot x^2) + \frac{1}{1+\delta} \cdot x
\]

As illustrated in Figure 3.10, the overall transfer speed of CSTP is about 30% higher than the overall transfer speed of parallel RBUDP streams; the maximum throughput of CSTP is about 20% higher than that of RBUDP.

![Figure 3.10: CSTP vs. RBUDP in parallel data transfer (X-axis: Number of Parallel Data Streams; Y-axis: Overall Data Transfer Speed in MB/s)](image)

**3.6 Conclusion**

We studied the problems which come along with large-scale data transfer/streaming, and present a solution, the CSTP protocol. Traditional point-to-point protocols are
not suitable for today’s cluster-to-cluster communication scenarios, used by scientific applications, and existing multi-to-multi-point protocols do not sufficiently address the synchronization issue of parallel data streaming and cannot tolerate burst packet loss. In this paper, we propose our new Cross-Stream Transfer Protocol (CSTP), which used cross-stream fountain code and pipe packets to tightly couple streams and help lossy streams to decode correctly and tolerate burst packet loss. Our experimental results show that the CSTP meets our expectation in terms of burst packet loss tolerance, decoding efficiency, communication overhead, and overall data throughput.

Numerous challenges remain to be addressed. Our future work will mainly look at the scalability issue of our CSTP algorithm. We also intend to add modules for reliable delivery to the current CSTP implementation.

In the next Chapter, we will focus on the scalability of Cross-Stream Coding.

3.7 Acknowledgements

Part of the content in this chapter has been published, in Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage. Jurgen Schulze and Thomas A. DeFanti co-authored the paper. The dissertation author was the primary investigator and author of this paper.

Part of the content in this chapter has been submitted to the journal of Future Generation Computer Systems, 2011. Jurgen Schulze and Thomas A. DeFanti co-authored the paper. The dissertation author was the primary investigator and author of this paper.
Chapter 4

Scalability of Cross-Stream Coding

4.1 Introduction

In this chapter we extend our new Cross-Stream (CS) coding approach to large numbers of parallel data streams. This approach requires that a parallel computer infrastructure is present at each end, with network connections between all nodes within each end point. There has to be an equal amount of sender nodes and receiver nodes. Each sender needs to have a network connection to at least one receiver, and vice versa. Our method can be applied to a large number of parallel data connections, even when packet loss is unevenly distributed across these connections. In previous chapters, we have experimented with the approach using up to 15 parallel data connections. As the number of connections increases further, the overhead associated with CS coding may also increase. From the analysis of CS coding in previous chapters, it is not obvious whether CS coding scales to a large number of data connections, because it is so far unclear how the overhead of CS coding will affect the scalability of the algorithm. When CS coding is done using the simpler approach from Chapter 3, the cost of decoding can be a linear function of the number of parallel streams. In the worst case, as in the example shown there, the communication cost will continue to increase significantly as the number of streams increases. There are also other types of overhead which may be factors that could prevent CS coding from scaling well.

Therefore, when we investigate the scalability of CS coding, we will primarily examine the type of overhead which might increase with the number of parallel streams.
It is obvious that some types of overhead are directly associated with the number of streams, for example the cost of synchronization, since synchronizing 100 streams will obviously incur a higher cost than synchronizing 10 streams. Although synchronization is not a priority of our research, it can be a limiting factor that might negatively affect the effectiveness of our CS coding approach. In this chapter, we present an empirical study of how well our synchronization approach works with large numbers of participating CPU nodes.

The communication cost caused by CS coding can greatly limit its scalability. Intuitively, it is natural to think that the coding complexity will be a function of the number of parallel streams. So when we consider a large number of streams, i.e., hundreds of parallel streams, we face the challenge of how to limit the increase of the communication cost as the number of streams increases, or a challenge about whether it is possible to keep the communication cost constant as the number of streams increases. We are going to investigate how we can keep this communication cost as constant and independent of the number of streams as possible.

The above issues are the driving factor for our deeper exploration of the CS coding mechanism. In this chapter, we will first explain what the synchronization issue is, and then focus on the communication cost issue in CS coding. We will explain the concept of communication cost, and start a detailed analysis of CS coding by analyzing the effectiveness of a single CS code. We will then prove a few theorems, stating the necessary and/or sufficient conditions of a successful CS coding mechanism under unevenly distributed code packet loss across parallel streams. Through our theorems, we will learn under what conditions the CS approach can recover all lost packets at the destination. And our theorem can yield feasible solutions using a linear programming approach, which we are going to explain in detail.

4.2 Synchronization

The synchronization in our Cross-Stream Coding approach is designed in a traditional way which has been applied to many similar parallel systems, for instance the StarCAVE [DDS+09] or SAGE [JRJ+06]. Our implementation is different than the
Table 4.1: Synchronization speed

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Speed (times/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>660</td>
</tr>
<tr>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>80</td>
<td>500</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
</tr>
</tbody>
</table>

above, but the principle is the same. We use a two-stage synchronization model to synchronize our parallel data streams.

Among the data streams, we designate one to be the head node, while all others will be worker nodes. Each node hosts exactly one CS sender or receiver.

Upon synchronization, the head node will:

1. Collect “SyncOK” messages from all the other nodes.

2. Send “SyncCFM” to other nodes once it has collected the “SyncOK” from all the other nodes, then proceed.

Upon synchronization, the worker nodes will:

1. Send a “SyncOK” to the head node.

2. Receive a “SyncCFM” from the head node, then proceed.

Although similar approaches have been commonly used in parallel data processing systems such as COVISE in the StarCAVE or SAGE on tiled display walls, they do not normally run on systems with more than a few dozen nodes. For example, in the StarCAVE, the synchronization happens for less than 20 nodes, while SAGE typically synchronizes up to around 30 nodes. To understand the speed of synchronizing larger numbers of nodes, we experimented with our algorithm on the FWGrid cluster [FWG] which has more than 100 nodes. The results of this experiment are listed in Table 4.1.

This experiment shows that our synchronization algorithm scales well up to 100 nodes, which means that our synchronization approach works well enough for such large numbers of nodes and does not require further optimization.
4.3 Cross-Stream Coding Communication Cost

Cross-Stream Coding communication cost constitutes of encoding cost and decoding cost. Although decoding complexity is significantly higher than encoding, the communication costs associated with them are similar. We will explain shortly.

4.3.1 What Incurs Communication Cost?

**Cross-Stream Code** Let each encoder independently encode their N original data packets into fountain code packets, generating N’ regular code packets. Then a Cross-Stream code packet, or a CS code, can be generated by XORing multiple regular code packets from multiple encoders. The regular fountain codes contained in a CS code are sent to/received from the peer encoders.

\[ Y = C_{i1} \oplus C_{i2} \oplus C_{i3} \oplus \ldots \oplus C_{iM} \]  

(4.1)

**Degree of a CS code** If Y is a CS code generated and held by stream A, and it is composed of one local regular fountain code, and \((M-1)\) regular fountain codes collected from other \((M-1)\) distinct streams, the degree of Y is defined as M. It is easy to tell that any CS code that does not meet the requirements above can be replaced by a better one.

As has been stated in previous chapters, communication cost is derived from encoding or decoding a CS code, because the CS code is composed of multiple regular code packets from multiple streams. For example, assuming Y is a CS code packet with degree M held by stream A, to encode Y, some of the components of Y must be retrieved from other streams through network communication.

4.3.2 Encoding Communication Cost

From previous sections, we know that since Y is a CS code in stream A, then \(C_{i1}, \ldots, C_{iM}\) contain one and only one local code packet of that stream. Without losing generalization, we rewrite Y as this, using \(M = 11\) as an example:
\[ Y = L_A \oplus C_{i1} \oplus C_{i2} \oplus C_{i3} \oplus \ldots \oplus C_{i10} \]

To encode \( Y \), encoder \( A \) needs to receive ten regular code packets from ten other encoders. Thus the communication cost that \( Y \) will incur is ten \cdot \text{sizeof(Regular Code Packet)}, or simply ten, as the size of code packets is a predefined constant value. A CS code with degree \( M \) will incur \((M - 1)\) units of communication cost.

If we have \( S \) streams, assuming we insert \( \delta \cdot N' \) CS code packets into the \( N' \) regular code packets in each stream, and \( d_i (2 \leq i \leq S) \) represent the percentage of CS code packets with degree \( i \), the overall communication in each of the encoders is upper-bounded by this formula:

\[
N' \cdot \delta \cdot \sum_{i=2}^{S} d_i \cdot (i - 1) \tag{4.2}
\]

where \( \sum_{i=2}^{S} d_i = 1 \).

*(Notes: \( \delta \) represents the overhead of CS coding, which needs to be a small number. More detail about \( \delta \) will be given later.)*

Encoding communication cost is precisely defined by formula 4.2. However, the coding details about how to choose \( d_i \) and how to choose \( \delta \) are actually determined by the requirements of the decoding procedure, and the encoding must be done efficiently to be feasible for the decoders to decode.

### 4.3.3 Decoding Communication Cost

Unlike the encoding procedure, decoding is not a deterministic procedure. In ideal cases when there is no code packet loss in any streams, all of the decoders can decode themselves solely by using the regular code packets they receive, and no CS codes will be needed to recover their original data. Therefore, no communication cost will be incurred. But in worse cases when there is considerable code loss, especially when it is unevenly distributed among the decoders, a large majority of the CS codes will be necessary for the lossy decoders to recover the original data. Consequently, a large number of code packets will be transferred between the decoders. In extreme cases, if too many code packets are missing, it might be impossible to fully recover the original data.
Under all of the above situations, the overall communication cost in each of the decoders cannot exceed the value defined in formula 4.2.

The parameters defined in formula 4.2 determine the communication cost of both the encoding and decoding procedures, and are the key to the scalability of the Cross-Stream Coding approach. In the next section, we will explore how to optimize these parameters.

### 4.4 Cross-Stream Coding Principle Analysis

We have stated earlier that formula 4.2 should be quantified to make the decoding process more effective. By effective, we mean the CS codes should be effectively leveraged to recover partial code loss in arbitrary decoders, when the partial code loss is unevenly distributed among them. And, we need to quantify the effectiveness.

In Chapter 3, we discussed that once the decoders have received the code packets, the decoding will start from the first stage, an independent self-decoding procedure invoked and run in all decoders in parallel. If after the first stage, any decoders have difficulty to recover all of their original data due to a high code loss rate, all decoders will enter the second decoding stage, in which the CS codes will be leveraged to help those decoders incapable of self-decoding to recover their original data. Although logically separated, in the implementation stage one and stage two are interleaved and CS codes will be processed and transferred as soon as they are available to improve the overall decoding speed.

The time behavior of this decoding procedure is non-deterministic. The code loss pattern cannot be predicted, neither can the percentage of decoders incapable of self-decoding. In our pattern loss analysis in Chapter 3 we made assumptions about these uncertainties. We assumed:

1. Most streams are able to recover their original data.
2. Data loss among streams are unevenly distributed, but the average packet loss percentage is relatively small.

The first assumption is based on an observation that in a parallel system only a
small percentage fails simultaneously; assumption two is based on experiments and real-world experience. Unevenly distributed data loss is much harder to repair comparing to evenly distributed data loss.

### 4.4.1 Formulation of The Problem

We use multiple parallel streams to transfer data between two distant locations. Each stream is composed of a data source, a data destination and a communication channel connecting them. Original raw data are decoded into redundant fountain codes at the sources, and are decoded at the destinations tolerating a certain percentage of code loss. Communication channels are unreliable and may cause potentially unevenly distributed code loss among streams during the transfer. The streams that suffer from high code loss cannot decode successfully solely with the regular fountain codes. So, we add additional Cross-Stream codes into each of the streams to enable the lossy streams to reconstruct their data with the help of other receiver nodes.

However, it is not straightforward how CS codes should be integrated with respect to the amount of CS codes and their stream degree distribution function.

The major challenges of the CS coding scheme are:

1. How to efficiently do CS coding to reduce unnecessary redundancy?
2. How to easily predict the recoverability given the code loss pattern and the actual codes arriving at destinations?
3. How to scale CS coding to large numbers of parallel streams?

In this chapter, we will respond to these challenges. We begin with the analysis of one CS code packet.

### 4.4.2 Effectiveness of A CS Code

We start from the analysis of the effectiveness of a CS code. After the first stage of decoding, all parallel decoders have completed self-decoding, and the percentage of successful recovery is $P$, while the percentage of unsuccessful recovery is $(1 - P)$. Then
we examine the effectiveness of an arbitrary CS code $Y$. $Y$ is a CS code if and only if it has been encoded as the “XOR of $M$ regular code packets from $M$ distinct streams”, which means $Y = C_{i1} \oplus C_{i2} \oplus C_{i3} \oplus \ldots \oplus C_{iM}$. $C_{ij}$ is a regular code packet from stream $P_{ij}$, for all $ij, (1 \leq j \leq M), ij \in [1 \ldots S]$. And also, for any pair of regular code packets $C_{im}$ and $C_{in}$ in $Y$, $im \neq in$.

[Note: This is neither a constraint nor an assumption, it is so because there is no need to include in a CS code more than one regular code packet from the same stream.]

For example, assuming $Y = C_{i1} \oplus C_{i2} \oplus C_{i3} \oplus \ldots \oplus C_{iM}$ is a CS code received by decoder $A$, and at the moment, all decoders except $P_{ij}$ have fully recovered their data, then all of the regular code packets in $Y$ except $C_{ij}$ are known to $A$, because those regular code packets are simply the XOR of recovered data packets from fully recovered decoders. Then $A$ can extract $C_{ij}$ using this formula: $C_{ij} = Y \oplus C_{i1} \oplus C_{i2} \oplus C_{i(j-1)} \oplus C_{i(j+1)} \oplus \ldots \oplus C_{iM}$. $C_{ij}$ is immediately sent to $P_{ij}$ as a new fountain code, and $Y$ is instantly effective.

4.4.3 Instant Effectiveness

Now, we introduce the first definition for the effectiveness of a CS code.

**Instant Effectiveness** The Instant Effectiveness($e$) of a CS code is defined as the possibility of the CS code being able to immediately generate a new regular fountain code packet for any individual decoder that has not yet fully recovered its original data and need help from peer decoders.

**Lemma 4.4.1.** Let $Y = C_{i1} \oplus C_{i2} \oplus C_{i3} \oplus \ldots \oplus C_{iM}$ be a CS code, then at any moment the possibility of $Y$ being an effective CS code is at least $M \cdot (1 - P) \cdot P^{(M-1)}$, where $M$ is the stream-degree of $Y$, and $P$ is the current percentage of decoders that have fully recovered their original data.

**Proof.** We first show counter examples where $Y$ is unable to help other decoders to recover their original data, when $P$ of the total number of decoders have decoded successfully. Evidently, if all regular codes $C_{i1} \ldots C_{iM}$ belong to streams that have already
successfully recovered their data, then $Y$ is useless; if more than two of the regular codes in $C_{i1}...C_{iM}$ were from streams which have not recovered yet, then $Y$ cannot be lever-aged either, because the unknown part in $Y$ is the XOR of two or more regular code packets, and there is no way to extract any of them from that combination so that no codes can be sent to any decoder to act as a new fountain code packet.

![Figure 4.1: Effectiveness of a CS code](image)

To quantify the instant effectiveness of a CS code, we analyze the effectiveness of $Y$ using an analogical example. If we view $C_{i1}...C_{iM}$ as $M$ black balls, and we view the percentages of recovered and un-recovered decoders as two containers whose sizes are $P$ and $(1 - P)$ respectively, we then randomly throw the $M$ balls into the two containers, as shown in Figure 4.1. For each ball, the chance of where it will land is proportional to the size of the containers: the chance of it being dropped into the container of size $P$ is $P$; the chance of it being dropped into the container of size $(1-P)$ is $(1-P)$. Since regular code packets $C_{i1}...C_{iM}$ are all randomly chosen from the $S$ streams, they can arbitrarily belong to the decoders that have fully decoded ($P$), or the decoders that have not $(1-P)$. The chance that any of them belongs to $P$ is equivalent to a ball dropped into the container with size $P$; the chance that any of them belongs to $(1-P)$ is equivalent to a ball dropped into the container with size $(1-P)$. Therefore, the possibility
that \((M-i)\) packets \(C_{i1}......C_{iM}\) are from the portion of recovered decoders and \(i\) packets of them are from the portion of un-recovered decoders is the same as the possibility that \((M-i)\) balls are dropped into the container of size \(P\) and \(i\) balls are dropped into the container of size \((1-P)\) in the analogical model.

For example, as shown in Figure 4.2, if all code packets of \(Y\) are from the decoders that have recovered, which is analogous to the situation where all \(M\) balls are dropped into the container of size \(P\) and no balls are dropped into the container of size \((1-P)\), then \(Y\) is not an effective CS code. The possibility of this situation is obviously \(P^M\). In this case, \(Y\) is useless as it will not be able to help any decoder that has not fully recovered its original data.

Similarly, as shown in Figure 4.3, if \(Y\) has two or more than two regular code packets which are from those decoders that have not recovered, which is analogous to the situation where there exist two or more than two balls which were dropped into the container of size \((1-P)\), then \(Y\) is not an effective CS code either. In this situation, \(Y\) can be useful in the future but is not instantly effective because more than two codes are unknown in \(Y\), and \(Y\) cannot extract any codes to generate useful code packets for the decoders that have not fully recovered.
As shown in Figure 4.4, the possibility of one of the balls being dropped into the container of size $(1-P)$ and $(M-1)$ balls being dropped into the container with size $P$ is $M \cdot (1-P) \cdot P^{M-1}$. Therefore, when $P$ of the decoders have completely decoded, the possibility of $Y$ being an instant effective CS code is $M \cdot (1-P) \cdot P^{M-1}$.

\[ e(M, P) = M \cdot (1-P) \cdot P^{(M-1)} \]

\[
\text{Corollary 4.4.2. The Instant Effectiveness of a CS code with degree } M \text{ is}
\]

\[
\text{Corollary 4.4.3. } e \text{ is not a monotonic function.}
\]

Proof. Let us give counter examples. We draw the curves of $e(M,P)$; $M = 2, 3, 4, 8, 20$. $P$ varies between 0.6 and 0.99. 0.6 is an arbitrary number used for demonstration purposes. The curves are drawn in Figure 4.5.
In Figure 4.5, five curves are shown. When M=2, e(M,P) is monotonically decreasing with P when P’s range is between 0.6 and 1; for a higher value of M, none of the curves are monotonic.

From these Instant Effectiveness curves of CS codes with different degrees (M), we can draw a few conclusions:

1. There is no magic number M that is the best in all situations.

   As shown in the plots, when P is a smaller value, the Instant Effectiveness favors a smaller value of M; when P is a larger value, the Instant Effectiveness function favors a bigger value. This can be intuitively explained using our analogical example: when you want to throw balls into two containers of moderate sizes, P and (1-P) respectively, the more balls you have, the greater the chances that you will throw more than two balls into the container with size (1-P), and make that an ineffective situation; when throwing balls into a large container of size P and a small container of size (1-P), you will need enough balls, meaning a larger M, to have a chance that one ball will fall into the small container.

2. Combination of CS codes might improve the overall effectiveness.
Figure 4.5: Instant Effectiveness of a CS code

When encoding a CS code, we can decide on its stream degree. We can also give different stream degrees to the CS codes. Since the stream degree of a CS code determines its effectiveness property, using combinations of CS codes might improve the overall effectiveness.

In summary, Instant Effectiveness quantifies the momentary effectiveness of a CS code, and it helps us understand how useful a CS code can be at every time point. Although, Instant Effectiveness is only a static measurement. Back to the original problem of parallel fountain code decoding across large number of streams, after self-decoding, most of the decoders are fully decoded, the remaining decoders need to leverage the CS codes to decode their original data. Even if all these decoders eventually can successfully decode, the whole process will not finish instantly but gradually instead.

The whole decoding procedure is a cycle: when there are effective CS codes, these codes can help more decoders to decode; when more decoders have successfully decoded themselves, more CS codes can become effective. As shown in Figure 4.6, this cycle stops when:

1. No more effective CS codes are available at a point in time; Or,
2. All decoders have successfully decoded themselves.

Clearly, Instant Effectiveness itself is not sufficient to address this dynamic procedure, as it cannot quantify the incremental effectiveness of CS codes when more and more decoders have successfully decoded. Thus, we need another tool to quantify this.

![Figure 4.6: Incremental decoding cycle using CS codes](image)

### 4.4.4 Incremental Effectiveness

Because of the instant effectiveness of CS codes, decoders that fall into the \((1-P)\) category will get extra fountain codes from other decoders, and then will be able to recover more of their original data. If the amount of extra codes is sufficient, they can fully decode their data, and the number of fully recovered decoders then will increase, which means \(P\) will increase. Now we need to know, when \(P\) increases, what will happen to the effectiveness of a CS code. If a CS code \(Y\) was an effective code, it has been already applied in the last step; if \(Y\) was not an effective code, it has a chance to become useful after \(P\) increases. We want to quantify the incremental effectiveness (ie) of \(Y\) with \(P\)'s increase.
Thus, we have the following definition.

**Incremental Effectiveness** We define the Incremental Effectiveness of a CS code as the possibility of its being turned into an effective code as \( P \) increases. When \( P \) increases as a result of the effectiveness of CS codes, new CS codes that were useless now can become effective. And Incremental Effectiveness quantifies the derivative of the incremental usefulness.

*Note: There is a possible misunderstanding when thinking of the Incremental Effectiveness. It is natural to think \( \text{ie}(M, P) = e'(M, P) = \lim_{\varepsilon \to 0} \left( \frac{e(M, P + \varepsilon) - e(M, P)}{\varepsilon} \right) \). But this is essentially wrong, because when \( P \) increases, \( Y \) has a possibility of changing from useless to useful, but also has a possibility of changing from useful to useless. The formula above measures both, which is not the right measurement we want to calculate.*

**Lemma 4.4.4.** Incremental Effectiveness is non-negative.

*Proof.* Different from Corollary 4.4.3, the definition of Incremental effectiveness only considers the possibility that a CS code turns into effective when \( P \) increases. There are four cases:

1. The code was useful, and is useless now: the Incremental Effectiveness of the code is 0.
2. The code was useless, and is useless now: the Incremental Effectiveness of the code is 0.
3. The code was useful, and is useful now: this is impossible, so the Incremental Effectiveness of the code is 0.
4. The code was useless, and is useful now: the incremental effectiveness of the code is positive.

**Lemma 4.4.5.** Incremental Effectiveness is a function of \( M \) and \( P \), and

\[
\text{ie}(M, P) = M \cdot (M - 1) \cdot (1 - P) \cdot P^{M-2}
\]

(4.4)
Proof. Again, we use the analogy example to complete the proof. Assuming that P increases to P+e, as shown in Figure 4.7; now we construct three containers the balls can be dropped into, whose sizes are P, e, and (1-P-e). CS code Y becomes useful after P increases if and only if Y satisfies the conditions drawn in Figure 4.7:

1. Only one ball is dropped into the container of size (1-P-e).
2. At least one ball is dropped into the container of size e.

The first condition guarantees that Y is now an effective CS code, the second condition guarantees that Y is not an effective CS code before P increases. It is easy to tell that these conditions are necessary and sufficient.

The possibility that Y meets the first condition is \( M \cdot (1 - P - e) \cdot (P + e)^{M-1} \); the possibility that Y meets the second condition is \( 1 - (\frac{P}{P+e})^{M-1} \), therefore the possibility that Y meets both requirements can be calculated as: \( M \cdot (1 - P - e) \cdot (P + e)^{M-1} \cdot (1 - (\frac{P}{P+e})^{M-1}) \).

It follows that

\[
\text{ie}(M, P) = \lim_{e \to 0} \frac{M \cdot (1 - P - e) \cdot (P + e)^{M-1} \cdot (1 - (\frac{P}{P+e})^{M-1})}{e} = M \cdot (M - 1) \cdot (1 - P) \cdot P^{M-2}
\]

Figure 4.8 shows \( \text{ie}(M, P) \) according to the same series of M values in Figure 4.5. Again, those curves exhibit very different properties. When \( M = 2 \), \( \text{ie}(M, P) = 2 \cdot (1 - P) \), which is a decreasing function of P. When M has a higher value, the curve will have a peak value for some P, and the peak location will shift to the right as M increases.

Incremental Effectiveness measures the positive change of the effectiveness of CS codes, and catches the dynamic property of them. Once we have the \( \text{ie}(M, P) \), we can calculate the Accumulative Effectiveness by integrating \( \text{ie}(M, P) \). The definition of Accumulative Effectiveness is given in the next section.
4.4.5 Accumulative Effectiveness

Let $P_0$ be the percentage of decoders which have fully decoded after the self-decoding stage. Starting from $P = P_0$, with the help of the instant effectiveness of CS code at point $P = P_0$, some decoders will become fully decoded and will make more CS codes effective, which can be measured by the incremental effectiveness function. As the decoding cycle (shown in Figure 4.6) repeatedly goes on, the incremental effectiveness of a CS code will accumulate as $P$ increases. We can measure the accumulative effectiveness of a CS code, and quantify the value for any given $M$ and $P$. This leads to the definition of Accumulative Effectiveness.

**Accumulative Effectiveness** Accumulative Effectiveness is defined as the accumulated effectiveness of a CS code from the initial state where $P_0$ of the decoders have finished decoding to the current state in which $P$ of the decoders have decoded with the assistance of CS codes. By definition, we have this equation:

$$AE(M, P, P_0) = e(M, P_0) + \int_{P_0}^{P} (ie(M, x) \cdot dx)$$

(4.5)
Figure 4.8: Incremental Effectiveness curves

Note: This equation assumes an infinite number of streams, and is a fair approximation as long as $S$ is large. Similar approximation methods are used in the rest of this chapter.

Corollary 4.4.6. $AE(M, P, P_0)$ is NOT the same as Instant Effectiveness $e(M, P)$.

Proof. It is obvious that their formulas are different.

But let us also give an intuitive explanation. $e(M, P)$ is the effectiveness of a CS code when $P$ of the decoders are decoded, but $AE(M, P, P_0)$ assumes that $P_0$ of the decoders have self-recovered, and that $(P - P_0)$ of the decoders have recovered by leveraging the effectiveness of CS codes.

Lemma 4.4.7. Accumulative Effective AE can be calculated using:

$$AE(M, P, P_0) = M \cdot P^{(M-1)} - (M - 1) \cdot P^M - P_0^M$$

Proof. Based on the definition of $AE(M, P, P_0)$,
\[ AE(M, P, P_0) = e(M, P_0) + \int_{P_0}^{P} i e(M, x) \cdot dx \]
\[ = M \cdot (1 - P_0) \cdot P_0^{M-1} + \int_{P_0}^{P} M \cdot (M - 1) \cdot (1 - x) \cdot x^{M-2} \cdot dx \]
\[ = M \cdot (1 - P_0) \cdot P_0^{M-1} + M \cdot (M - 1) \int_{P_0}^{P} (x^{M-2} - x^{M-1}) \cdot dx \]
\[ = M \cdot (1 - P_0) \cdot P_0^{M-1} + (M \cdot x^{M-1} - (M - 1) \cdot x^{M}) |_{P_0}^{P} \]
\[ = M \cdot P^{M-1} - (M - 1) \cdot P^{M} - P_0^{M} \]

**Corollary 4.4.8.** \( AE(M, 1, P_0) = 1 - P_0^M \)

\[ AE(M, P_0, P_0) = e(M, P_0) \]

\[ AE'(M, P) = ie(M, P) \]

**Proof.** By Lemma 4.4.7.

**Corollary 4.4.9.** \( AE \) is a non-decreasing function.

**Proof.** By Corollary 4.4.9, \( AE'(M, P) = ie(M, P) \geq 0 \), so \( AE \) is a non-decreasing function.

**Corollary 4.4.10.** When \( P_0 \) is the percentage of decoders which have decoded independently, for any CS code \( Y \) having unknown parts in its composition, \( Y \) will eventually be effective at some point as long as the decoding stage will last until all parallel decoders have successfully decoded themselves.

**Proof.** As the decoding moves along, more composing codes of \( Y \) are known, until only one regular code in \( Y \) is unknown, and that code will become an effective regular code for the decoder it belongs to.

In Figure 4.9, we plot the curves of Accumulative Effectiveness using the same series of values of \( M \) as in Figure 4.5 and Figure 4.8. For CS codes with small stream degree \( M \), the AE curve suggests that they are more useful when \( P \) is small; for CS codes
with larger stream degree $M$, the AE curve suggests that they are unlikely to be effective at the beginning, but will be useful when $P$ reaches further to the right. Both types of CS codes are important. The first type of CS codes initiates the whole procedure and pushes $P$ to a higher value, and the second type of CS codes continues the procedure when the first type is exhausted.

![Accumulative Effectiveness curves](image)

**Figure 4.9**: Accumulative Effectiveness curves

In summary, our analysis results in three important principles:

1. When $P$ is small, we need CS codes with small stream degree $M$ to initiate the CS decoding.
2. We need CS codes with medium stream degrees to proceed stably.
3. CS codes with high stream degree are very effective towards the end of the decoding procedure.

Intuitively, reading from the curves, we want to mix CS codes with a range of different stream degrees, from smaller values to larger values. However, in practice, to
quantify them, for example, to determine the distribution function of stream degrees, we cannot combine them arbitrarily. The selection of of CS codes with various stream degrees and their weights is related to the objective parallel data loss pattern. In the following sections, we will assume some pattern of parallel code loss, and based on these assumptions, we will manage to prove L’s three theorems, and determine the cross-stream distribution function using Linear Programming guided by those theorems.

Before we continue, we need to introduce a few auxiliary concepts first.

### 4.4.6 Auxiliary Concepts

When we describe a CS code as effective, we mean that it is effective for “some decoder” which has not fully decoded. This decoder can be any one in the (1-P) portion of the parallel decoders. Those decoders equally share the effectiveness, meaning the effectiveness can be calculated by dividing the effectiveness by (1-P).

**Average Stream Effectiveness** Average Stream Effectiveness (ASE) is the average effectiveness with which all streams belonging to the (1-P) portion can benefit from the effectiveness of CS codes.

*Notes: Average Stream Effectiveness can be directly applied to \( e(M,P) \), \( ie(M,P) \), but cannot be directly applied to \( AE(M,P,P_0) \).*

**Lemma 4.4.11.**

\[
APE_{e(M,P)} = \frac{e(M,P)}{1 - P} = M \cdot P^{M-1}
\]

(4.6)

\[
APE_{ie(M,P)} = \frac{ie(M,P)}{1 - P} = M \cdot (M - 1) \cdot P^{M-2}
\]

(4.7)

*Proof.* Proven by definition.

**Corollary 4.4.12.** \( APE_{ie(M,P)} = APE'_{e(M,P)} \)
Proof. By Lemma 4.4.11.

We mentioned parallel code packet loss patterns in our earlier analysis. A pattern is a statistical property rather than an individual property. For example, if we only have two streams and in one situation their loss rates are 5% and 10%, while in another their loss rates are 10% and 5%. These two apparently different situations share the same code loss patterns. Our study will not differentiate between the details of each individual stream’s code loss, but will focus on the data recovery for the parallel loss pattern. Since we assume an unevenly distributed code loss pattern among streams, some streams may lose very little code or none at all, some may lose a big portion of the code, and some may be somewhere in-between. The characteristics of the parallel data loss pattern is determined by the aggregation of the data loss statistics.

**Stream locator S(P)** Once we know the parallel code loss pattern, each stream will have a unique position in the sorted list, so that it can be exclusively located by the code loss rate of that stream. If more than two streams have the same data loss rate, they will still have a unique location in the sorted list based on loss rate, arbitrarily decided by the sorting method.

Each stream can be specified by its relative position in the list, and we can use S(P) to specify a stream, where P is the percentile of that stream. \( S\left( \frac{1}{3} \right) \) is the stream whose loss rate is the lowest; \( S(1) \) is the stream whose loss rate is the highest.

**Parallel Stream Code Packet Loss Pattern f(P)** We use f(P) to define the statistics of data loss rates, where P varies between 0 to 1. f(P) is the data loss rate of S(P). f(P) is evidently an increasing function. \( f\left( \frac{1}{3} \right) \) is the lowest data loss rate, and f(1) is the highest data loss rate. For convenience, f(0) is defined to be 0.

We studied some loss patterns in Chapter 3. The patterns discussed there are evenly-distributed, linear, polynomial, random, etc. Based on experiments, polynomial and random models are more realistic models. Although we use a random model as the example in the following sections, our algorithms do not rely on that, meaning the algorithms do not only work for random or polynomial models. If an application has a
specific \( f(P) \), the algorithm can be used to find an optimal solution for it. Theoretically, the parallel loss pattern is merely an input parameter to the algorithms.

Nevertheless, by studying the random/polynomial model, the most versatile class of problems will be addressed in general. In extreme situations, there are always ad-hoc approaches. For instance, if we want to tolerate a single stream failure of 100\%, an additional, hot-spare data stream will recover the data. If we want to address evenly distributed loss patterns, the best way is to use traditional Forward Error Correction code. Our algorithms focus on loss patterns in-between the above.

In Figure 4.10, we draw a curve of a random division function \( f(P) \). We set the average code loss rate to 5\%. In the curve, the most lossy stream loses 33\% of its codes.

![Random Division Model, Average 5%](image)

**Figure 4.10**: An example of \( f(P) \)

Although \( f(P) \) is not known previously, we can assume an \( f(P) \) to be our objective parallel loss pattern, and work out a solution to recover this objective \( f(P) \). Therefore, the same solution solves any actual pattern that can be covered by the objective \( f(P) \). Let \( \alpha \) be the average code loss rate of \( f(P) \), we assume that code loss lower than \( \alpha \) in each of the streams should be tolerated in the self-decoding phase. Based on this
principle, using Figure 4.10 as the example, Figure 4.11 illustrates the situations after
the self-decoding phase is completed. All streams which have less data loss (less than
5%) can successfully recover themselves and are located in the green zone; all other
streams which have higher code loss rates (higher than 5%) cannot decode their original
data and are located in the red zone. The figure shows that in this example about 65% of the streams can recover their original data, which is

\[ P_0 = 0.65, \quad f(P_0) = 0.05. \]

Therefore, the challenge is to help the streams projected to the red part to recover
their lost code packets. The help is carried through CS codes, and the amount should
be sufficient to make up the code loss beyond the tolerable portion in each of those
streams. In Figure 4.12, we use yellow to mark off the portion that fountain code is able
to tolerate, and the remaining red area is the portion CS codes have to recover.

The challenge is: the lossy streams are not at the same level in terms of how
much help they need. For instance, S(0.66) only needs a very small amount of extra
codes, while S(1) will need 28% of extra codes to succeed. How can our Cross-Stream
coding efficiently address those different requirements and avoid wasting bandwidth

**Figure 4.11**: Small portion of packet loss can be self-recovered.
caused by creating unnecessary codes beyond those requirements? We introduce a series of theorems of Effective Parallel Cross-Stream Decoding, or L’s theorems. We are going to prove the three theorems L1, L2 and L3. L1 and L2 are based on the assumptions we discussed earlier, and L3 makes some of those assumptions unnecessary.

### 4.5 L1 Theorem of Scalable Cross-Stream Parallel Decoding

#### 4.5.1 L1 Theorem

**Theorem 4.5.1 (L1:Absolute Cover Theorem ).** Let $\varepsilon$ be the erasure rate of native fountain code used in CS coding, $f(P)$ be the objective parallel code loss pattern, and $\alpha$ be the average code loss rate of $f(P)$, then the total number of regular code in each stream is $N' = \frac{N(1+\varepsilon)}{1-\alpha}$, which is the amount that can tolerate $\alpha$ code loss in each of the streams.
71

\( P_0 \) is the percentile where \( f(P_0) = \alpha \). Let \( N' \cdot \delta \) be the number of CS codes which are integrated in each of the streams, \( N'' = (1 + \delta) \cdot N' \) is, therefore, the number of all codes in each stream. Based on these conditions, the original data of all streams can be recovered if the following conditions are satisfied:

1. \[
\frac{\sum_{i=2}^{S} \delta \cdot (1 - \alpha) \cdot d_i \cdot AE(i, P_0)}{1 - P_0} \geq f(P_0 + \frac{1}{S}) - \alpha, \text{where} f(P_0) = \alpha
\]

2. For any \( P, P_0 + \frac{1}{S} \leq P < 1, \)
\[
\frac{\sum_{i=2}^{S} \delta \cdot (1 - \alpha) \cdot d_i \cdot AE'(i, P)}{1 - P} \geq f'(P), \text{where} f(P_0) = \alpha
\]

![Figure 4.13: The next stream is \( S(P + \frac{1}{S}) \)](image)

**Proof.** First, the streams decode their original data solely using regular code packets. Since CS codes are not used in this self decoding phase, the performance of self decoding is only related to the overhead of regular fountain code. For each data stream, as long as its code loss rate is no more than \( \alpha \), the number of regular codes is sufficient to
decode their original data. So after the self decoding phase, \( P_0 \) of all streams will have successfully recovered their original data, where \( f(P_0) = \alpha \). Figure 4.11 illustrates an example with an \( f(P) \) curve with \( \alpha = 5\% \).

1. After the self decoding phase, the streams are from \( S(\frac{1}{3}) \) to \( S(P_0) \), where the variable \( S \) is the total number of parallel streams. The first stream that cannot yet be decoded is \( S(P_0 + \frac{1}{3}) \); the amount of regular codes it needs is, by definition is \( N' \cdot (f(P_0 + \frac{1}{3}) - \alpha) \). From Lemma 4.4.1, the Instant Effectiveness of a CS code is \( \sum_{i=2}^{S}(d_i \cdot AE(i, P_0)) \). Since the total number of CS codes is \( S \cdot N' \cdot \delta \cdot (1 - \alpha) \), the total number of effective codes is \( S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S}(d_i \cdot AE(i, P_0)) \). Because of the random generation of CS codes, the effective codes are evenly shared by \( (1 - P_0) \cdot S \) streams, so that each stream will receive \( \frac{N' \cdot \delta \cdot (1 - \alpha) \sum_{i=2}^{S}(d_i \cdot AE(i, P_0))}{1 - P_0} \) extra fountain codes. For \( S(P_0 + \frac{1}{3}) \) to get enough extra codes to decode, this number should be at least the number required by it. That is, \( \frac{\sum_{i=2}^{S} \delta \cdot (1 - \alpha) \cdot d_i \cdot AE(i, P_0)}{1 - P_0} \geq f(P_0 + \frac{1}{3}) - \alpha \), which immediately leads to:

\[
\frac{\sum_{i=2}^{S} \delta \cdot (1 - \alpha) \cdot d_i \cdot AE(i, P_0)}{1 - P_0} \geq f(P_0 + \frac{1}{3}) - \alpha, \text{ where } f(P_0) = \alpha
\]

2. For any \( P' \), \( P_0 + \frac{1}{3} \leq P < 1 \), using induction, assuming the streams from \( S(\frac{1}{3}) \) to \( S(P) \) have decoded all of their original data, and the next closest stream is thus \( S(P + \frac{1}{3}) \), which is halted and needs extra codes to proceed.

We start the analysis from right before \( S(P) \) is being decoded. At that moment, the effectiveness \( S(P) \) and all later streams have equally gained is \( S(P + \frac{1}{3}) \), which is the accumulated effectiveness \( AE(P) \), highlighted in yellow in Figure 4.12. The level of the yellow portion is at least as high as shown in Figure 4.13, because it is the smallest level needed by \( S(P) \) to decode itself, as required by the induction assumption.

In addition, after \( S(P) \) is decoded, the newly available effective fountain codes are quantified by the following formula: \( S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S}(d_i \cdot (AE(i, P) - AE(i, P - \frac{1}{3}))) \), and the number of codes each stream from \( S(P + \frac{1}{3}) \) to \( S(1) \) will get equals

\[
\frac{S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S}(d_i \cdot (AE(i, P) - AE(i, P - \frac{1}{3})))}{(1 - P) \cdot S}
\]
which equals
\[
\frac{N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot (AE(i,P) - AE(i,P - \frac{1}{S}))}{(1 - P)}
\]

If this amount is equal or greater than the required amount by \( S(P + \frac{1}{S}) \), which is obviously \( N' \cdot (f(P + \frac{1}{S}) - f(P)) \) as indicated in Figure 4.13, \( S(P + \frac{1}{S}) \) will be able to fully decode itself, and the whole procedure will move on to the next stream. Thus, we have the following condition:

\[
\frac{N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot (AE(i,P) - AE(i,P - \frac{1}{S}))}{(1 - P)} \geq N' \cdot (f(P + \frac{1}{S}) - f(P))
\]

This implies the following:

\[
\frac{\delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot (AE(i,P) - AE(i,P - \frac{1}{S}))}{(1 - P)} \geq f(P + \frac{1}{S}) - f(P)
\]

\[
\Rightarrow \frac{\delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot (AE(i,P) - AE(i,P - \frac{1}{S}))}{(1 - P)} \cdot \frac{1}{S} \geq \frac{f(P + \frac{1}{S}) - f(P)}{\frac{1}{S}}
\]

\[
\Rightarrow \frac{\delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot AE'(i,P))}{(1 - P)} \geq f'(P)
\]

This is the second condition of the theorem.

We proved that if condition 1 is satisfied, the decoding procedure can proceed from \( S(P_0) \) to \( S(P_0 + \frac{1}{S}) \). If condition 2 is satisfied, the decoding procedure can proceed from any \( S(P) \), \( P \geq P_0 + \frac{1}{S} \), to \( S(P + \frac{1}{S}) \) until \( P = 1 \).

Theoretically, both conditions are only determined by the shape of curve \( f(P) \), but not to the number of parallel streams. \( S \) is the sampling rate of \( f(P) \). If \( f(P) \) remains the same, a solution yielded from the theorem with some sufficiently big value of \( S \) will also work for larger values of \( S \). So theorem L1 will lead to practical solutions which scale to a large number of parallel streams.

\[\square\]
4.5.2 Experimenting with Theorem L1

The conditions in theorem L1 are very strong. They are sufficient but not necessary. This means that the solution yielded from L1 is practically inefficient.

![f(P) curve](image)

Figure 4.14: f(P)

1. Objective function f(P)

We first generate an objective f(P) function. By setting S=100, targeting an average data loss rate of 5%, using the random division model mentioned in Chapter 3, we generated a f(P) curve as shown in Figure 4.14. Assuming the individual code packet loss tolerance rate $\alpha = 5\%$, $P_0$ can be estimated as 60%. In Figure 4.14, we plot the heuristic f(P) curve, the upper one is the complete f(P) $0 < P \leq 1$, the lower one is the partial f(P), $P_0 \leq P \leq 1$.

2. Find a feasible solution using Linear Programming

We want to use the L1 theorem to find the right values of $\delta$ and $d_i$. 
In L1, f(P) and \( P_0 \) are assumed or summarized similar to what is shown in Figure 4.14. Given the f(P) and estimated \( P_0 \), the variables in L1 theorem is \( \delta \) and \( d_i \). We use linear programming to determine those variables. By replacing \( d_i+1 \) with \( x_i \), 1 \( \leq i \leq S - 1 \), the conditions of linear programming are:

\[
\begin{align*}
    x_i & \geq 0; 1 \leq i \leq (S - 1) \\
    \sum_{i=1}^{S-1} x_i & = 1 \\
    A \cdot x & \geq f^T_A \\
    f_A & = \{ f(P_0 + \frac{1}{S}) - \alpha, f'(P_0 + \frac{j}{S}), 2 \leq j \leq (1 - P_0) \cdot S \} \\
\end{align*}
\]

The first row of A is:

\[
\{ \delta \cdot (1 - \alpha) \cdot (i + 1) \cdot P_0^{1 \leq i \leq (S - 1)} \}
\]

The \( j_{th}, (2 \leq j \leq (1 - P_0) \cdot S) \) row of A is:

\[
\{ \delta \cdot (1 - \alpha) \cdot i \cdot (i + 1) \cdot (P_0 + \frac{j - 1}{S})^{1 \leq i \leq (S - 1)} \}
\]

\[
CommCost = \sum_{i=1}^{S-1} i \cdot x_i
\]

Once we have derived a solution from linear programming, we can verifying the solution by plotting \( A \cdot x \) and \( f_A \), the x-axis is \([P_0, 1]\), the y-axis is the values in vectors \( A \cdot x \) and \( f_A \). We call the curve plotted using \( f_A \) the objective curve, or OBJ(P); we call the curve plotted using \( A \cdot x \) the solution curve, or L1(P).

We use the f(P) curve shown in Figure 4.14, the average code loss rate in f(P) is \( \alpha \), which equals to 5%. Then we set \( \delta \) to 15.1\%, the lowest value that can yield a solution, and plot both OBJ(P) and L1(P) in Figure 4.15.

Using theorem L1 and linear programming methods, we are able to find feasible solutions for a scalable coding scheme. However, L1 enforces unnecessarily strict requirements which make the solution much more costly. It requires that the slope of the Cross-Stream degree function exceeds the slope of f(P). This leads us to introduce our next theorem of scalable CS coding.
4.6 L2 Theorem of Scalable Cross-Stream Parallel Decoding

Theorem L1 will yield a feasible solution, but with high overhead and low efficiency. Starting with the experimental results of L1, we first made a few aggressive assumptions and tested them, which led to our 2nd theorem of Scalable CS Coding.

4.6.1 L2 Theorem

Theorem 4.6.1 (L2: Integration Cover Theorem). Let $\varepsilon$ be the erasure rate of native fountain code used in CS coding, $f(P)$ be the objective parallel code loss pattern, and $\alpha$ the average code loss rate of $f(P)$; then the total number of regular code in each stream is

$$N' = \frac{N(1+\varepsilon)}{1-\alpha},$$

which is the amount which can tolerate $\alpha$ code loss in each of the streams. $P_b$ is the percentile where $f(P_b) = \alpha$. Let $N' \cdot \delta$ be the number of CS codes which are integrated in each of the streams. $N'' = (1+\delta) \cdot N'$ is, therefore, the number of all codes in each stream. Based on these conditions, the original data of all of the streams can be recovered, if and only if the following conditions are satisfied:
For all $P, P_0 \leq P < 1$,

$$\delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1} \geq f(P + \frac{1}{S}) - \alpha$$

, where $\delta$ is the percentage of CS codes integrated into each of the streams. $d_i (2 \leq i \leq S)$ are the coefficients of the Cross-Stream coding distribution function.

Proof. We assume stream $S(P | P \leq P_0)$ have decoded successfully after the self decoding phase.

1. When $P' = P_0 + \frac{1}{3}$, the total number of instantly effective CS codes is

$$S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot e(i, P_0)$$

which equals to

$$S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot (1 - P_0) \cdot P_0^{i-1}$$

These codes will be evenly shared by $(1 - P_0) \cdot S$ streams which have not decoded, so the number of extra code packets stream $S(P_0 + \frac{1}{3})$ will get is:

$$\frac{S \cdot N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot (1 - P_0) \cdot P_0^{i-1}}{(1 - P_0) \cdot S} = N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1}$$

Stream $S(P_0 + \frac{1}{3})$ can fully decode itself if and only if it loses no more than $\alpha$ of its code packets. Since $S(P_0 + \frac{1}{3})$ actually loses $N' \cdot f(P_0 + \frac{1}{3})$ regular code packets, it need at least $N' \cdot (f(P_0 + \frac{1}{3}) - \alpha)$ extra effective codes from CS codes. This requirement leads to this formula:

$$N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1} \geq N' \cdot (f(P_0 + \frac{1}{3}) - \alpha)$$

$$\implies \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1} \geq f(P_0 + \frac{1}{3}) - \alpha$$
2. Using induction, for any $P', P_0 + \frac{1}{3} \leq P' < 1$, at the moment when all $S(P|P \leq P')$ have been successfully decoded, and all $S(P|P > P')$ have not been successfully decoded, $S(P' + \frac{1}{3})$ is the next stream that requires minimal amount of extra codes to decode. The amount of codes that $S(P' + \frac{1}{3})$ has acquired since the beginning equals to

$$\frac{N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot e(i, P_0)}{1 - P_0} + N' \cdot \delta \cdot (1 - \alpha) \cdot \int_{P_0}^{P'} \frac{\sum_{i=2}^{S} d_i \cdot i e(i, P)}{1 - P} \cdot dP.$$  

The first part of the formula above are the Instantly Effective codes which $S(P' + \frac{1}{3})$ can receive when $P = P_0$. The second part of the formula is the amount of Incrementally Effective codes which $S(P' + \frac{1}{3})$ can receive when the decoding procedure goes on from $P = P_0$ to $P = P'$. Working on this formula, we have:

$$\frac{N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot e(i, P_0)}{1 - P_0} + N' \cdot \delta \cdot (1 - \alpha) \cdot \int_{P_0}^{P'} \frac{\sum_{i=2}^{S} d_i \cdot i e(i, P)}{1 - P} \cdot dP$$

$$= N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot (P_0^{i-1} + \int_{P_0}^{P'} (i - 1) \cdot P^{i-2} \cdot dP)$$

$$= N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{(i-1)}$$

$S(P' + \frac{1}{3})$ needs $N' \cdot (f(P' + \frac{1}{3}) - \alpha)$ code packets to decode. Thus, the following condition is the sufficient and necessary condition for it to decode:

$$N' \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{(i-1)} \geq N' \cdot (f(P' + \frac{1}{3}) - \alpha)$$

$$\implies \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{(i-1)} \geq f(P' + \frac{1}{3}) - \alpha$$

By summarizing 1 and 2, we proved that the necessary and sufficient condition to decode the original data packets in all streams is:

for all $P, P_0 \leq P < 1$,

$$\delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1} \geq f(P + \frac{1}{3}) - \alpha$$
Theorem L2 also suggests that a solution satisfying the required conditions can be scaled to large numbers of parallel streams. If a solution works for 100 parallel streams, the same solution also works for 1000 parallel streams as long as the \( f(P) \) remains the same.

### 4.6.2 Experimenting with L2 Theorem

1. **Objective function \( f(P) \)**

   We use the same objective \( f(P) \) function as L1 in Figure 4.14.

2. **Using L2 to find the optimal CS coding distribution function with Linear Programming**

   The L2 theorem clearly states the conditions for a feasible solution of how to design the CS code stream stream composition. To use that theorem to get a practical solution with linear programming, we need to use some integer value of \( S \), the number of parallel streams, as the sampling rate of \( f(P) \). By choosing a reasonably large number, for instance \( S=100 \), we will have sufficient granularity to characterize \( f(P) \) and draw a general solution applicable to a larger value of \( S \).

   There can be many feasible solutions, and the optimal solution is the one with minimized overhead: the Cross-Stream decoding communication cost.

   By replacing \( d_{i+1} \) with \( x_i \), \( 1 \leq i \leq S - 1 \), the conditions of linear programming are:
\[
x_i \geq 0; 1 \leq i \leq (S - 1)
\]
\[
\sum_{i=1}^{S-1} x_i = 1
\]
\[
A \cdot \delta \cdot (1 - \alpha) \cdot x \geq f_A^T - \alpha
\]
\[
f_{A}[j] = f(P_0 + \frac{j}{S}), 1 \leq j \leq (1 - P_0) \cdot S
\]
\[
A[i,j] = (i + 1) \cdot (P_0 + \frac{j - 1}{S})^i, 1 \leq i \leq (S - 1), 1 \leq j \leq (1 - P_0) \cdot S
\]
\[
\text{CommCost} = \sum_{i=1}^{S-1} i \cdot x_i
\]

To solve the linear programming (LP) problem, we need to choose a value for \(\delta\) first. It is easy to tell that if \(\delta\) is very high, then the formula in L2 will be easy to satisfy. Without losing generalization, we start by setting \(\delta = \alpha\) in the linear programming problem and experiment with different \(\delta\) values to find the relationship between \(\delta\) and the optimal CS coding degree distribution function.

The objective function of the LP is \(\text{CommCost}\).

By setting the CS codes percentage to \(\delta = 5\%\), we run the LP to find the optimal values of \(x_i\). By replacing \(x_i\) with \(d_{i+1}\), we get the optimal Cross-Stream coding degree distribution function \(D_5(x) = \frac{1}{1 + \delta} \cdot x + \frac{\delta}{1 + \delta} \cdot (0.9888 \cdot x^5 + 0.0112 \cdot x^6)\).

This degree distribution function suggests that 99% of the CS codes have a stream degree of 5, and about 1% of them have a stream degree of 6. Given that, the average code degree is \(0.9888 \cdot 5 + 0.0112 \cdot 6 = 5.01\). The maximum CS decoding communication cost for each of the decoding streams is \((5.01 - 1) \cdot \delta = 20\%\).

This means: when integrating 5% CS codes into each stream and using \(D_5(x)\), the CS codes can effectively help all streams recover their code loss as long as their code loss pattern can be covered by \(f(P)\).

In theorem L2, our targeting function \(\text{OBJ}(P)\) is \(f(P + \frac{1}{5}) - \alpha, P_0 \leq P < 1\), the solution curve using L2 is \(L2(P) = \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1}\). We plot both curves in Figure 4.16, and the figure confirms that the conditions in L2 are indeed satisfied.
In addition, we can compare the solutions found by Theorem L2 and L1. The method is to use the solution of Theorem L1 as the input to generate L2(P). Thus, we can clearly understand that the properties of L1. Figure 4.17 illustrates the optimal solution found by Theorem L1. We confirm that L1 not only requires higher space overhead, but also is inefficient because the L2(P) is very far away from the OBJ(P).

3. Space Cost Vs. Communication Cost

Using theorem L2 and linear programming, we can find the best feasible CS coding degree distribution function for any given $\delta$. In Figure 4.16, we saw an example where $\delta = 5\%$. But if we change the value of $\delta$, we will get different solutions from linear programming.

Higher values of $\delta$ will apparently make the LP conditions easier and so that will yield an “easier” solution; lower values of $\delta$ will make it more difficult and so it will yield a “harder” solution. However, to make the linear programming solvable,
\( \delta \) cannot be arbitrarily small. In this example, the minimal value of \( \delta \) is approximately 3\%. When \( \delta = 3\% \), the resulting degree distribution function found by linear programming is \( D_S(x) = \frac{1}{1+\delta} \cdot x + \frac{\delta}{1+\delta} \cdot (0.935 \cdot x^9 + 0.075 \cdot x^{10}) \). The average degree of \( D_S(x) \) equals to 9.06, and the communication cost is \((9.06 - 1) \cdot \delta = 24\%\). Figure 4.18 illustrates this solution. Intuitively, as \( \delta \) gets smaller, the linear programming has more constraints, and the resulting L2(P) curve is then closer to the objective curve OBJ(P).

In summary, \( \delta \) and \( D_S(x) \) represent two types of overhead. \( \delta \) specifies the extra space needed to save or transfer the CS codes, while the average degree of \( D_S(x) \), or \( AD(D_S(x)) \), specifies the CS coding complexity. \( \delta \cdot (AD(D_S(x)) - 1) \) quantifies the coding communication cost in the worst case. We use two functions to identify these two types of overhead: \( Space(\delta) \) and \( Comm(\delta) \), and

**Figure 4.17**: The L2(P) curve of a L1 solution (\( \delta = 15.1\% \))
Figure 4.18: L2(P) vs. OBJ(P) ($\delta = 3\%$)

\[
\begin{align*}
\text{Space}(\delta) &= \delta \\
\text{Comm}(\delta) &= \delta \cdot (AD(D_S(x)) - 1)
\end{align*}
\]  

(4.8)

where $D_S(x)$ is the optimal degree distribution function derived from L2 with a given $\delta$.

In a straightforward analysis, $Space(\delta)$ and $Comm(\delta)$ may conflict with each other. When lowering $\delta$, the space cost decreases, but $D_S(x)$ will be more complex so that the communication cost might increase. Thus we draw the relationship between the space cost and the communication cost in Figure 4.19. Increasing $\delta$ will gradually reduce the overall communication cost until $\delta$ reaches a certain high value. In this example, when $\delta = 12\%$, the optimal $D_S(x)$ will be 

\[
D_S(x) = \frac{1}{1+\delta} \cdot x + \frac{\delta}{1+\delta} \cdot x^2.
\]

After that moment, the communication cost will increase linearly with $\delta$ as $D_S(x)$ cannot be further simplified.
Sacrificing space to gain the benefit of low communication cost is not always desirable for applications, as space cost is fixed overhead that cannot be avoided, but communication cost is a overhead that only happens when necessary. For instance, if no code loss is observed, then the communication cost can be very low. In another words, communication cost is a very variable overhead.

High space cost also brings with it a large amount of waste of effectiveness of CS codes, and pulls down the efficiency of CS codes. In Figure 4.18 and Figure 4.16, we observed two different situations. In Figure 4.16, the L2(P) curve is farther departed from the OBJ(P) curve, and in Figure 4.18, the L2(P) is more tightly coupled with the OBJ(P) curve. This difference suggests a difference of CS coding efficiency in the two situations. So we introduce a new term called Cross-Stream Coding Efficiency.

4. Efficiency of CS Coding

**Cross-Stream Coding Efficiency** Given a solution that satisfies the conditions
in theorem L2, Cross-Stream Coding Efficiency (CCE) is defined as the required effectiveness by \( f(P) \) over the available effectiveness provided by that solution. The necessary effectiveness to recover all data is 
\[
N' \cdot \sum_{P_0, S+1}^S (f \left( \frac{i}{S} \right) - \alpha)
\]
and the available effectiveness by the solution is
\[
S \cdot N' \cdot \sum_{i=2}^S d_i \cdot AE(P_0, 1, i).
\]
So,
\[
CCE(D_S(x)) = \frac{N' \cdot \sum_{P_0, S+1}^S (f \left( \frac{i}{S} \right) - \alpha)}{S \cdot N' \cdot \sum_{i=2}^S d_i \cdot AE(P_0, 1, i)} \tag{4.9}
\]

**Corollary 4.6.2.** Based on theorem L2, if LP can find a solution \( D_S(x) \) that satisfies \( \sum_{i=2}^S d_i \cdot i \cdot P^{i-1} = f(P + \frac{1}{S}) - \alpha \), for all \( P, P_0 \leq P < 1 \), then \( CCE(D_S(x)) \approx 1 \). 

**Proof.** The number of effective codes needed can be represented as below:

\[
N' \cdot \sum_{P_0, S+1}^S (f \left( \frac{i}{S} \right) - \alpha)
\]

\[
\begin{align*}
&= N' \cdot \sum_{i=1}^{(1-P_0)S} (f(P_0 + \frac{i}{S}) - \alpha) \\
&= N' \cdot S \cdot \sum_{i=1}^{(1-P_0)S} (f(P_0 + \frac{i}{S}) - \alpha) \cdot \frac{1}{S} \\
&= N' \cdot S \cdot \sum_{P=P_0}^{1-\frac{1}{S}} (f(P + \frac{1}{S}) - \alpha) \cdot \frac{1}{S}
\end{align*}
\]

In the formula above, \( \sum_{P=P_0}^{1-\frac{1}{S}} (f(P + \frac{1}{S}) - \alpha) \) can be seen as a sampling of the continuous function \( \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P^{i-1} \), because \( \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P^{i-1} = f(P + \frac{1}{S}) - \alpha \), for all \( P, P_0 \leq P < 1 \). So that \( \sum_{P=P_0}^{1-\frac{1}{S}} (f(P + \frac{1}{S}) - \alpha) \cdot \frac{1}{S} \) is an approximation of the area of the curve \( \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P^{i-1} \), \( P_0 \leq P \leq 1 \), which equals to,
\[
\int_{P=P_0}^1 \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P^{i-1}
\]
\[
= \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \int_{P=P_0}^1 i \cdot P^{i-1}
\]
\[
= \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot P^i |_{P=P_0}^{1}
\]
\[
= \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot (1 - P_0^i)
\]

Thus, the number of effective code packets needed is approximately \(N' \cdot S \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot (1 - P_0^i)\).

On the other hand, the total number of available effective CS codes is \(N' \cdot S \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot AE(P_0, 1, i)\), which also equals to \(N' \cdot S \cdot \delta \cdot (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot (1 - P_0^i)\).

Therefore, \(CCE(D(x)) \approx 1\).

Increasing \(\delta\) not only increases the space cost, but also reduces the coding efficiency of CS code, meaning that too much useless information is stored/transferred to recover the objective code loss pattern. In Figure 4.20, when increasing \(\delta\), CCE (the number of used CS code/the number of effective CS code) will decrease. Meanwhile, the overall code usage efficiency (the number of used CS code compared to the number of all CS code) will also decrease.

### 4.7 L3 Theorem of Scalable CS Coding

L2 improves L1. But still, it requires an estimation of \(P_0\) and start the analysis from \(P_0\). From the proof of L2, we got some insight into removing the definition of \(P_0\) and start our analysis from the beginning. As a result, we came up with Theorem L3, which is a unified scalability theorem of CS coding, which eliminates all definitions of intermediate variables and some assumptions.
4.7.1 L3 Theorem

**Theorem 4.7.1** (L3: Unified Scalability Theorem). *Let the erasure rate of fountain code be \( \varepsilon \), and let \( \gamma \) be the overhead of all code packets \( (\gamma = \frac{N'}{N} - 1) \). Let \( f(P) \) be the objective code loss pattern curve, \( \alpha \) be the average code loss rate, and let \( d_1...d_S \) be the percentages of code with stream degree 1...S, stream degree 1 characterizing regular codes. The original data packets in all streams can be recovered if and only if:*

\[
(1 + \gamma) \cdot (d_1 \cdot (1 - f(P))) + (1 - \alpha) \cdot \sum_{i=2}^{S} (d_i \cdot i \cdot (P - \frac{1}{i})^{i-1}) \geq 1 + \varepsilon \quad (4.10)
\]

**Proof.** We prove the theorem in two steps.

1. **First Stream:** \( P = \frac{1}{3} \)

   For the first stream with the lowest code loss rate, \( P = \frac{1}{3} \), it can only use regular fountain code to recover its original data. The number of regular fountain codes
it holds equals to $N'' \cdot d_1 \cdot (1 - f(1/S))$, which should be at least $N \cdot (1 + \varepsilon)$. This means, $(1 + \gamma) \cdot d_1 \cdot (1 - f(1/S)) \geq 1 + \varepsilon$.

2. For all of the following Streams: $S(P')$

For any $P' \leq P < 1$, at the moment when all $S(P|P \leq P')$ have successfully decoded, and all $S(P|P > P')$ have not successfully decoded, $S(P' + \frac{1}{S})$ is the next process that requires a minimal amount of data to decode. The amount of codes that $S(P' + \frac{1}{S})$ has acquired since the beginning equals to

$$N'' \cdot d_1 \cdot (1 - f(P + \frac{1}{S})) + N'' \cdot (1 - \alpha) \cdot \int_0^{P'} \sum_{i=2}^S d_i \cdot ie(i,P) \cdot \frac{1}{1 - P} \cdot dP.$$

The first part in the above formula are the regular code packets that $S(P' + \frac{1}{S})$ received, the second part is the amount of Incremental Effective codes that $S(P' + \frac{1}{S})$ can receive when the decoding procedure goes on from $P = 0$ to $P = P'$. Working on this formula, we have

$$N'' \cdot d_1 \cdot (1 - f(P' + \frac{1}{S})) + N'' \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot \int_0^{P'} i \cdot (i - 1) \cdot P^{i-2} \cdot dP$$

$$= N'' \cdot d_1 \cdot (1 - f(P' + \frac{1}{S})) + N'' \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P'(i-1)$$

Similarly, $S(P' + \frac{1}{S})$ needs $N \cdot (1 + \varepsilon)$ regular code packets in total to decode. Thus, the necessary and sufficient condition for it to decode is:

$$N'' \cdot d_1 \cdot (1 - f(P' + \frac{1}{S})) + N'' \cdot (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P'(i-1) \geq N \cdot (1 + \varepsilon)$$

$$\Rightarrow N \cdot (1 + \gamma) \cdot (d_1 \cdot (1 - f(P' + \frac{1}{S})) + (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P'(i-1)) \geq N \cdot (1 + \varepsilon)$$

$$\Rightarrow (1 + \gamma) \cdot (d_1 \cdot (1 - f(P' + \frac{1}{S})) + (1 - \alpha) \cdot \sum_{i=2}^S d_i \cdot i \cdot P'(i-1)) \geq 1 + \varepsilon$$
This proves that for any $P', \frac{1}{S} \leq P' < 1$, $S(P' + \frac{1}{S})$ can decode itself if and only if

$$(1 + \gamma) \cdot (d_1 \cdot (1 - f(P' + \frac{1}{S})) + (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot P'^{(i-1)}) \geq 1 + \varepsilon$$

which means, by substituting $P' + \frac{1}{S}$ with $P$: for any $P, \frac{1}{S} < P \leq 1$, $S(P)$ can decode itself if and only if

$$(1 + \gamma) \cdot (d_1 \cdot (1 - f(P)) + (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot (P - \frac{1}{S})^{(i-1)}) \geq (1 + \varepsilon)$$

By summarizing (1) and (2), we proved that, for any $P, 0 < P \leq 1$, $S(P)$ can decode itself if and only if

$$(1 + \gamma) \cdot (d_1 \cdot (1 - f(P)) + (1 - \alpha) \cdot \sum_{i=2}^{S} d_i \cdot i \cdot (P - \frac{1}{S})^{(i-1)}) \geq (1 + \varepsilon)$$

4.7.2 Experimenting with Theorem L3

1. Using L3 to find the optimal CS coding distribution function with Linear Programming

In equation 4.10, $\gamma$ and $d_1 \cdot d_S$ are variables, $\varepsilon$ is known, $\alpha$ and $f(P)$ are objective constants. In our experiments, we use $\varepsilon = 5\%$, and $\alpha = 5\%$, and assume $f(P)$ is the curve shown in Figure 4.14. For any given $\gamma$, the communication cost, $(1 + \gamma) \cdot \sum_{i=2}^{S} d_i \cdot (i - 1)$, is the objective function, whose value needs to be minimized. LP will find the optimal $D_S(x) = d_1 \cdot x + d_2 \cdot x^2 + \ldots + d_S \cdot x^S$, which can minimize the communication cost.

We still use linear programming to find the optimal solution. The conditions for
the LP are:
\[ d_i \geq 0; 1 \leq i \leq S \]
\[ \sum_{i=1}^{S} d_i = 1 \]
\[ A \cdot d \geq 1 + \varepsilon \]

Based on equation 4.10, the \( j \)th row of A is:
\[ \{(1 + \gamma) \cdot (1 - f(i/S)), (1 + \gamma) \cdot (1 - \alpha) \cdot i \cdot (\frac{j-1}{S})^{(i-1)}, 2 \leq i \leq S\}, 1 \leq j \leq S \]

\[ \text{CommCost} = (1 + \gamma) \cdot \sum_{i=2}^{S} (i - 1) \cdot d_i \]

\( \gamma \) is the amount of space overhead the users are willing to pay for. We first choose a fairly large value for \( \gamma \), and then study the effect of different \( \gamma \) values.

Assuming the user can tolerate 20% code overhead, including regular and CS code overhead, \( \gamma = \frac{N''}{N} - 1 \) is then 20%. Linear programming finds the best solution as below:

\[ D_S(x) = 87.5\% \cdot x + 10.6\% \cdot x^2 + 1.9\% \cdot x^3 \]

\( D_S(x) \) is the optimal way to do CS coding when \( \gamma \) equals 20%, meaning the number of code packets is 120% the number of original data packets. Among those code packets, 87.5% are regular code packets, 10.6% are CS codes with stream degree 2, and 1.9% are CS codes with stream degree 3.

2. Space Overhead VS. Communication Overhead

To easily understand the overhead of the solution, we look at Figure 4.21. The horizontal line is the required number of code packets to decode for \( S(P) \), and the curve \( L3(P) \) is the actual number of regular code packets \( S(P) \) can have, including regular code packets generated from CS code, from the solution given by Linear Programming. In the figure, the gap between the two lines is significant.

This figure will look different when we change the value of \( \gamma \). As \( \gamma \) decreases, the \( D_S(x) \) solution found by Linear programming will be more complex. If \( \gamma \) is lower than a certain threshold, a feasible solution can no longer be found. For the given \( f(P) \), the lowest value of \( \gamma \) that can lead to a feasible solution is around 11.4%. 
Figure 4.21: Using L3 to find the best feasible solution ($\gamma = 20\%$)

$$D_S(x) = 94.22\% \cdot x + 2.89\% \cdot x^2 + 2.47\% \cdot x^8 + 0.26\% \cdot x^{17} + 0.15\% \cdot x^{18}$$

The trade-off between space overhead and communication overhead depends on the characteristics of the application. And any metric containing both factors can be easily optimized as well.

3. Code Efficiency

In L3, code efficiency shares the same definition with L2 in principle: the necessary code effectiveness over the actual available code effectiveness. Because L3 does not use the same assumption as L2, the representation of Code Efficiency in L3 is slightly different. In L3, we define the efficiency as: $$\frac{1+\varepsilon}{\int_0^1 L3(P) \, dP}.$$ Similar to corollary 4.6.2, as $L3(P)$ converges to $BAR(P)$, the code efficiency converges to $100\%$. 

4.8 Experimenting with other $f(P)$ curves

In our theorems, the objective curve $f(P)$ is only an input parameter, which are used to generate the optimal solution. For any given objective $f(P)$ and space overhead, our algorithm is able to generate the best possible stream degree distribution function which can recover the packet loss pattern specified by $f(P)$. Here are a few examples.

4.8.1 Linear Loss Pattern

Let the $f(P)$ be a linear pattern as shown in Figure 4.23.

The best solution derived from L3 is $D_S(x) = 95.32\% \cdot x + 4.68\% \cdot x^2$. Space overhead associated with this solution is 10.8%, and communication overhead is 5.38%. The coding efficiency is about 99.9%. As shown in Figure 4.24, the L3(P) and Bar(P) are almost overlapped.

From the figure, we can draw a conclusion: the best solution to recover linear loss pattern only uses CS codes with stream degree 2.
**Figure 4.23:** Linear loss pattern (X-axis: P; Y-axis: f(P))

**Figure 4.24:** Recovery of linear loss pattern
4.8.2 Stepped Appearance Loss Pattern

Another untypical but interesting loss pattern is shown in Figure 4.25.

Using this objective loss pattern, the best solution derived from L3 is \( D_5(x) = 93.51\% \cdot x + 3.16\% \cdot x^3 + 1.43\% \cdot x^{15} + 1.9\% \cdot x^{16} \). The space overhead associated with this solution is 14\%, the communication overhead is 62\%, and the efficiency of the code is 96\%.

The OBJ(P) ad BAR(P) are drawn in Figure 4.26.

4.8.3 Comprehensive Metrics

The three measurements, space overhead, communication overhead and code efficiency, can be arbitrarily combined to form comprehensive metrics. And our algorithms can optimize those metrics in a straightforward way.
4.9 Emulation on FWGrid

We have experimented with our theorems in FWGrid, an advanced large-scale computer cluster in the Department of Computer Science & Engineering at the University of California, San Diego. The computers are essentially configured with Dual 1.6Ghz-3.2Ghz AMD processors and 1Gb Ethernet. Details of server and network configuration of FWGrid can be found in [FWG]. We run our emulation with 100 physical machines in FWGRID, and all our emulation are based on solutions derived from L3 Theorem.

We use the example \( f(P) \) in Figure 4.27. An optimal solution is \( D_5(x) = 94.05\% \cdot x + 3.47\% \cdot x^2 + 2.48\% \cdot x^7 \), and is drawn in Figure 4.28. In our emulation on FWGrid, we use this solution as our CS coding stream degree distribution function, expecting all data loss will be recovered. In reality, all streams are able to recover their original data, and the number of effective codes in each of the streams are shown in Figure 4.29.

In Figure 4.29, we observe that the actual numbers of effective codes in all the streams are almost the same after the Cross-Stream decoding, no matter how much data they lose at the beginning. This verifies that CS coding can effectively balance the level
Figure 4.27: Emulated loss pattern on FWGrid

Figure 4.28: An optimal solution to recover loss pattern on FWGrid
Figure 4.29: The actual emulation results (The pink line draws the number of final available codes in each of the streams.)

received data in different streams, as well as pipes can do to the water stream model illustrated in Figure 3.2.

Note: In Figure 4.29, the goal is to prove that \( f(P) \) is recoverable with \( D_S(x) \). The goal is NOT to prove that the pink line matches the blue one. The figure shows an evenly distributed result out of an unevenly distributed loss pattern.

4.10 Conclusion

In this Chapter, we study the scalability of CS coding. The most important basis is the analysis of CS code effectiveness. The study shows that CS with different stream degrees also exhibits different properties, in terms of how their Instant and Incremental Effectiveness vary as the percentage of fully decoded streams increases.

For any given \( f(P) \), the L theorems clearly define the conditions of a feasible method of doing CS coding, and using linear programming, we can always find the best feasible solution to tolerate that parallel loss pattern. And the solution is not related to
the number of parallel streams, but is only determined by the objective \( f(P) \). Therefore, those optimal solutions are in fact scalable; the Cross-Stream Coding is scalable.

4.11 Acknowledgements

The content in this chapter will be submitted to a conference or journal. The dissertation author will be the primary investigator and author of this paper. Co-authors of this paper have not been determined yet.
Chapter 5

CX: Using High-Speed Networks to Create A Workflow-based Peta-scale Distributed Storage Platform

5.1 Introduction

The Academy of Motion Picture Arts and Sciences (AMPAS) report “The Digital Dilemma” describes the issues caused by the rapid increase of storage requirements for long-term preservation and access of high quality digital media content. As one of the research communities focusing on very high quality digital content, CineGrid addresses these issues by building a global-scale distributed storage platform suitable for handling high quality digital media, which we call CineGrid Exchange (CX). Today, CX connects seven universities and research laboratories in five countries, managing 400TB of storage, of which 250TB are dedicated to CineGrid. All of these sites are interconnected through a 10Gbps dedicated optical network. The CX distributed repository holds digital motion pictures at HD, 2K and 4K resolutions; digital still images; and digital audio in various formats. The goals of the CX are: 1) Providing a 10Gbps interconnected distributed platform for the CineGrid community to study digital content related issues, e.g., digital archiving, the movie production process, and network transfer/streaming protocols 2) Building a tool with which people can securely store, easily
share and transfer very high definition digital content worldwide for exhibition and real-time collaboration; 3) Automating digital policies through middleware and metadata management. In this publication, we introduce the architecture of the CX, resources managed by the CX and the implementation of the first series of CX management policies using the iRODS programmable middleware. We evaluate the first phase of CX platform implementation. We show that CX has the potential to be a reliable and scalable digital management system.

CineGrid is a research community with the mission “To build an interdisciplinary community that is focused on the research, development, and demonstration of networked collaborative tools to enable the production, use, preservation and exchange of very-high-quality digital media over photonic networks.” Members of CineGrid are a mix of post-production facilities, media arts schools, research universities, scientific laboratories, and hardware/software developers around the world connected by up to 10 Gbps networks. Since 2005, CineGrid members have conducted pioneering experiments in digital media production and post-production, network streaming delivery, exhibition, and remote collaboration. These experiments created media assets that CineGrid members wanted to access over time, which meant they had to be stored somewhere, managed to insure access and preservation, and transferred upon request among members scattered around the world. The first CineGrid Exchange (CX) nodes at UCSD/Calit2, UvA and Keio/DMC were established on an ad-hoc basis to fulfill this requirement. But over time, as the number and size of CX nodes has increased, and the number and variety of digital media assets in the CX has grown, the CineGrid community has seen a need to systematically combine integrate existing CX resources in order to provide a scalable solution for the storage of digital assets.

Just as distributed rendering architectures have been adopted by cinema post-production facilities to deliver visual effects shots on tight schedules, the trends in cloud computing (private and public) for virtualization of servers, storage and high-speed network infrastructure can be adopted for distributed digital content creation, distribution, library and archiving services. CX is a pioneering effort to establish a global-scale networked test bed that can be used to experiment with pre-commercial technologies and prototype collaborative workflows. At the same time, CX fulfills CineGrid’s own re-
requirements for secure access to the organization’s terabytes of digital media content by replicating assets in geographically distributed repositories connected by persistent 10 Gbps networks.

With support from AMPAS and contributions from other CineGrid members, the CX development project was started in 2009 to design and implement a multi-layer open source digital media asset management system. This is the first large scale, distributed global storage implementation designed to handle digital motion picture materials at the highest quality and for the investigation of issues related to digital preservation storage networks.

The three major goals of the CX are: 1) Providing a dedicated 1Gbps-10Gbps interconnected distributed platform for the CineGrid community to study digital content related issues, e.g., digital archiving, the movie production process, and network transfer/streaming protocols; 2) Building a tool with which people can securely store, seamlessly stream, share and transfer very high quality digital content across the world; 3) Automating digital policies through middleware and metadata management, so people can define flexible content ingestion, digestion, replication, and deletion processes.

In Section II of this publication, we will discuss previous related work; in Section III, we will describe the three-layer architecture of the CineGrid Exchange, in Section IV, we will talk about the CX policies and workflows. In Section V, we show the user interfaces in the CX, and some experimental results and experiences. Finally, we will conclude and suggest future work in Section VI.

5.2 CX Architecture And Hierarchy

The CineGrid Exchange project intends to create persistent hardware and network configurations enabling a reliable distributed storage platform dedicated to the CineGrid community. This platform serves as a test-bed that continually evolves to fit the participants’ research requirements.

CX aims to logically combine physically distributed storage systems at a number of locations, and to manage these resources as if they were a single storage system. Setting up a worldwide distributed storage system involves a high level of international
cooperation to provision storage capacity, network links and technical support at every participating site.

The CineGrid Exchange was first instantiated at three locations Calit2 at the University of California San Diego, Keio University Research Institute for Digital Media and Content in Tokyo Japan, and the University of Amsterdam in The Netherlands. (Figure 5.1a) The three sites are interconnected using 10 GigE links provided by CineGrid network members who are part of the Global Lambda Integrated Facility (GLIF) and other research and educational networks around the world. Over time, more research institutes have joined the CineGrid Exchange, contributing storage and human resources to CX development. By the end of 2009, as shown in Figure 5.1b the CX has seven participants on three continents, interconnected by a 10 Gbps network. These participants provide the infrastructure for the CineGrid Exchange.

### 5.2.1 CX Infrastructure

The CineGrid Exchange infrastructure consists of the following components.

1. Storage subsystems

#### Table 5.1: CX nodes and their capacity

<table>
<thead>
<tr>
<th>CX Node Site</th>
<th>Storage Type</th>
<th>CX Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calit2/UCSD, San Diego, USA</td>
<td>Three Sun Thumpers (x4540)</td>
<td>66TB</td>
</tr>
<tr>
<td>UvA, Amsterdam, Holland</td>
<td>Sun Thumper (x4540)</td>
<td>30TB</td>
</tr>
<tr>
<td>EVL/UCI, Chicago, USA</td>
<td>RAID Array</td>
<td>10TB</td>
</tr>
<tr>
<td>DMC/Keio, Tokyo, Japan</td>
<td>RAID Array</td>
<td>8TB</td>
</tr>
<tr>
<td>CSENet, Prague, Czech Republic</td>
<td>Sun Thumper (x4540)</td>
<td>48TB</td>
</tr>
<tr>
<td>Ryerson U, Toronto, Canada</td>
<td>Two Sun Thumpers (x4540)</td>
<td>57TB</td>
</tr>
<tr>
<td>AMPAS, Los Angeles, USA</td>
<td>Sun Thumper (x4540)</td>
<td>24TB</td>
</tr>
</tbody>
</table>

A CX storage subsystem, also called a Content Repository, can be any type of storage system including LTO, RAID Array, or storage server. Equipment used for CX storage range from high speed Sun Fire X4540 Server (Thumper) to storage arrays connected to a Linux PC. The storage resources at CX nodes as of Jan 2010
(a) The CX’s three initial sites: San Diego, Tokyo and Amsterdam, which are connected via 10Gbps optical networks.

(b) By the end of 2009, the CX has included seven sites globally: Tokyo, San Diego, Los Angeles, Chicago, Toronto, Amsterdam and Prague. They are connected by 10Gbps optical networks.

**Figure 5.1:** Map of the CX sites
are shown in Table I. The total capacity of the CX expanded from 50 TB to 250 TB in 2009 and is still growing. Additional active sites in the USA, Brazil and Australia are planned for the first quarter of 2010. To address the need for offline storage resources, a LTO (Linear tape-open) storage/library solution is planned to provide CX long-term bulk storage capability in 2010.

2. Network bandwidth

Using DWDM (Dense Wavelength-division Multiplexing) technology, 1Gbps-10Gbps optical networks have been adopted by the global research community. CineGrid Exchange’s network infrastructure includes many high-speed advanced network resources (CaveWave, CiscoWave, StarLight, TransLight, JGN2, etc.) that are part of the GLIF. We will discuss later in this publication whether the 10Gbps, which are used in the CX may be over-provisioned.

3. Input/output Portal

Very high quality digital media content is often created using specialized media devices like 4K or higher resolution cameras. Very high definition content also needs a means to be displayed. As stated in Section II, specialized input/output devices (4K cameras, 4K projectors driven by a NTT codec, an OptIPortal or the StarCave driven by cluster computers capable of displaying very high definition digital content) can be considered an extension of CX resources and overall infrastructure.

4. Computing resources

It is expected that eventually user of the CX will also request computational tasks: image processing, compression/decompression, transcoding, format conversion, and encoding/decoding for secure network distribution. Therefore, clusters of general-purpose computers are also critical resources for the CX.

5. Cluster storage

Several general-purpose PC clusters are deployed as part of the CX infrastructure, serving as rendering nodes for display walls or virtual reality systems with powerful CPUs and GPUs. They can be used as computational resources, or as
distributed storage in the CX. For instance, a 30-node cluster with 1TB hard drives
in each node can be configured as a 30TB shared storage system. Special access
protocols, i.e. CSTP [LSD09], RobuSTore, have been developed to leverage these
low-cost, high-performance parallel storage systems.

All the above resources describe the CineGrid Exchange infrastructure, which
comprises the foundation layer for the CX. They support middleware layer de-
ployed on top of this foundation. Those resources are logically independent to
each other, if without using any upper level software to integrate them. The up-
per level software in the CX is primarily built on top of iRODS, including extra
components customized for the CX. They form the middleware level of the CX.

5.2.2 CX Middleware

CX middleware is an integration of tools and components, including distributed
data grid software, metadata management software, scheduler, workflow engine, net-
work protocols, etc.

1. iRODS: the Integrated Rule-Oriented Data System

iRODS plays a significant role in the CineGrid Exchange middleware layer. It
organizes distributed data into a shareable collection, allowing the user to view
files stored at multiple locations as a single logical collection. In the CX, each
site runs an iRODS server, which manages the local resource. iRODS servers can
exchange information directly. State information is stored in a centralized iRODS
CATalog (iCAT) server. Given a request from a client, the iCAT metadata cata-
log is accessed to authenticate the user and validate authorization. The location
containing the desired file is identified and the request is forwarded to the remote
storage location. A distributed rule engine located at the remote storage location
applies local policies that control functions such as redaction, metadata extrac-
tion, replication, and retention. Once a job is assigned to one iRODS server, an
iRODS agent is spawned locally in response to the request. For example, if an
iRODS client in Tokyo wants to retrieve files from the storage system located in
San Diego, the iRODS server running in San Diego will receive a request and
spawn a new instance of an iRODS Agent to handle that request and send back the files to the requesting client in Tokyo.

One key feature of iRODS is its programmability, which is implemented through the iRODS API as rules. Rules in iRODS are critical to CX because it makes CX more than just a decentralized FTP server, but also an intelligent distributed storage platform. (See Section IV for further details).

2. CX Catalog And Metadata Management

Today, indexing digital content poses many challenges, as extracting information from images or videos usually requires human intervention. Often, the availability of metadata is the only option to index, search and manage digital content. In the CX, iRODS uses a centralized database (iCAT) to manage file metadata, indexing the location, the directory structure, file names, creation dates, checksums, etc. Through the “imeta” command, iRODS provides the option to associate additional, user-defined metadata called Attribute-Value-Unit (AVU) metadata with individual files and directories inside of iRODS to support metadata searches.

However, the CX’s requirements on metadata-based searches go beyond what iRODS can offer. Therefore, we intend to integrate a more powerful metadata management tool within CX called CollectiveAccess [CA]. CollectiveAccess is open source software that provides a highly configurable cataloguing tool and a web-based user interface for museums, digital archives, and digital collections. As part of the CX project in 2009, CollectiveAccess has been enhanced to handle cinema and other high quality media assets. And an interface between CollectiveAccess and iRODS has been designed so that CX will be able to use CollectiveAccess as the “front end” of its iRODS-managed distributed repository in the next phase of implementation of the CX.

3. CX Management Components

When iRODS and CollectiveAccess are integrated in 2010, CineGrid Exchange development will have covered many of the basic goals of a distributed storage platform for high quality digital media. But there are still un-addressed challenges. These challenges require addition management components to be added
to the middleware layer of the CX. We can implement these components using the iRODS programmable interfaces, and improve the overall system’s intelligence and configurability. Although implementing all of these sophisticated rule-based components is not the goal of the first phase, it is very important that preliminary steps are compatible with future upgrades. Generally, we start with simple implementations, and will refine them as we move forward.

(a) Policy and Workflow Management The implementation of policy management component is an important factor in digital content management. It determines the workflows of how the data are ingested, distributed, archived, etc., as well as ensuring the digital content processes are complete and no steps are missed. The current CX policies and workflows are somewhat flexible as the system requirements are evolving. The first set of policies have been implemented and deployed, and we will discuss them in details in Section IV.

(b) Resource Management Resources in the CX are heterogeneous and will likely always remain so. As a result, we will need a resource management component to match resources to requests. Finding the right resources for a particular request not only requires the full knowledge of all resource capacities and performances, but also the understanding of how they are connected to the CX infrastructure and to the requesters. We expect that resource management will continue to require a great deal of careful thought and planning.

(c) Other Management Components Some of the management modules presently under consideration are: a job scheduler to schedule parallel jobs; fast parallel file transfer/streaming protocols to utilize network bandwidth most efficiently; computational components to perform compression/ decompression, encoding, /decoding and transcoding; a way to monitor the distributed system for overall reliability and fault tolerance, etc. Reporting/auditing has also been identified by the user community as important.

At present, basic policies and workflows have been implemented to ensure CX operability. Future research will aim to improve and expand on the cur-
rent rudimentary implementations. Where it makes sense to add middleware automata, functions currently managed by people will be automated. Because CX is intended to also serve as an experimental testbed for CineGrid members, it is also anticipated that external modules may need to be added in the future.

CX Middleware is largely based on iRODS, and will use CollectiveAccess as a front-end to iRODS metadata management scheme to allow more complete metadata indexing and user-friendly searches. The iRODS rules engine provides the programming interface for CX policies to support CX applications. The Collective Access application provides the user interface for media catalog browsing.

4. CX applications: Ingest, Distribution, Streaming, Real-Time Processing, Search, etc.

From the very beginning, the CX has been driven by applications such as ultra high quality video streaming. As mentioned before, previous demonstrations were successful, but were arduous and temporary. And once the demonstrations were over, the created infrastructure was taken down. A central goal of the CX is to support applications consistently by defining workflows and providing management functionality on a high reliable platform.

In summary, the CX is designed as a three-level hierarchical structure, as illustrated in Figure 5.2. At the lowest level is the infrastructure, network and storage resources linked together by lambda networks; the middle level is iRODS, metadata management tools (CollectiveAccess) and other management components that manage the content repositories and implement policies to support CX applications, which represent the top level in this model.

5.3 CX Policies And Workflows

The CineGrid Exchange Working Group defines policies which are then expressed as workflows to be implemented using iRODS rules. From the storage plat-
form perspective, supporting workflows as part of the storage system functionality is fairly novel, compared to a traditional storage system which mainly supports POSIX I/O operations like fopen, fread, fwrite, fclose, and fseek. The definition of the workflows in CX is largely based on real-world, digital content workflows. Although most workflows still need human assistance, the goal of CX is to make them as automatic as possible. Automation reduces the chance for human error or oversight and lessens the administrative burden of archive maintenance.

Content in the CX repository must be securely stored and distributed, and accurately retrieved. And once accepted into the CX collections, content should be preserved until a decision is made to purge from the collection. In addition, asset use is constrained by copyright considerations. Permission to use CX content varies widely. Though CX assets are limited to use by the CineGrid membership only, content is also classified for use based on member type. In some cases, an additional layer of constraint is imposed based on Contributor restrictions for intended use of the content.

It is widely accepted that metadata is also critical to digital content preservation
due to the technical difficulties in automatic indexing of digital content; however, people are reluctant to create metadata manually, compared to their enthusiasm in creating the content. So, it is critical to incorporate a workflow that mandates metadata generation at the time of content ingest into the CX. Cataloguing, searching and referencing digital media content without sufficient metadata is near to impossible, and far more taxing than simply storing or transferring files within a storage system.

Therefore, the CX policies are made to guarantee the safety, security, accuracy and manageability of digital media assets in the CX collection while making them easily accessible to community members. While these policies may seem straightforward at first glance, it was not easy to define them, especially because we want sufficient details to automate using iRODS-based rules. The content should be stored as multiple copies in a distributed manner, so that even if one site experiences a catastrophic failure, there are still identical copies around elsewhere, a digital preservation strategy first explored in the LOCKSS Program [LOC]. Periodical data verification is required to ensure that the archived content has not changed. Our policies avoid the accidental deletion of content by implementing a strict deletion protocol. After many refinements by the CineGrid Exchange Working Group, the first set of CX policies has now been defined. These policies are the most commonly used processes in the operation of the CineGrid Exchange. Two of the most important workflows are examined here.

5.3.1 CX Ingest Workflow

To put content into the CX, the content holder first submits an application to the (human) CX curator with a preliminary description of the material, including proxy and resource requirements. The CX curator reviews the application and responds after checking related information, e.g., usage permission and ownership. If the curator decides that it is okay to include the material, the content holder completes metadata entry into the cataloging system via a web form. If the content holder has a fast network connection to the CX, or operates a CX node of their own, they are ready to proceed. Otherwise, they must first ship their content on a hard drive to one of the established CX nodes and connect the hard drive to the CX. This requirement stems from the fact that the CX resides not on the public Internet, but is part of a limited access research
network.

For security reasons, CX node operators are not allowed direct access to the CX repositories. Each node operator is assigned a dropBox under his home directory within the CX, which is also managed by the CX middleware, but with appropriate user-level permissions. The content will be transferred into the node operator’s home dropBox, and will be checksummed. If necessary, files with mismatched checksums will be re-transferred until all checksum tests pass. At this time, automated iRODS processes take over. After the upload is completed, an internal workflow is activated through iRODS rules, which works like a system call in a UNIX/Linux system. The workflow will do the following things in this sequence:

1. Switch the mode of the operator to super-user of the CX.

2. Use the resource-matching algorithm to identify two additional storage resources from all resources in the CX. The resource-matching algorithm in the CX randomly selects two more resources from the resource collection, which includes all the available resources in the CX. More advanced algorithms are under development, which will consider other variables such as remaining storage space, storage quota, and other features supported by the latest release of iRODS.

3. Replicate two copies to the two identified resources and perform a checksum operation (It has been extensively discussed in the digital preservation community, and people agree that 3-copy is a mind-peace number that should be preserved for the digital contents.) Files are sequentially copied to three locations. With RBUDP integrated into our implementation, we have not felt this be a major bottleneck. However, for a long-term goal, we will consider to pipeline the process by overlapping multiple file transfers for the same sequential of files and checksum operations.

4. Move the files from the content holder’s home dropBox into the user’s corresponding directory in the repository.

5. Complete the process and register the files into CX database.

6. For all copies of each file, set the right access permissions.
7. A notification email is sent to the CX curator and content holder confirming successful ingest into the CX.

Figure 5.3: The CX ingest workflow (the portion of the workflow with a dark background is automated using CX middleware)

Guided by the content deletion policy, a non-delete rule will automatically be associated with the new directory so that it cannot be removed through any unauthorized interface or commands. By the content preservation policy, a periodical auditing rule is associated with the directory to do a bi-monthly auditing of the files in the directory.

Figure 5.3 illustrates the ingest workflow. The first few steps require human assistance, but can be partially automated through email/web applications; the portion of the workflow with a dark background is automated using CX middleware.
5.3.2 CX Distribution Workflow

At present, to retrieve protected content from the CX, the requester contacts the CX curator with their requests, specifying the name of the data object, data type (compressed version, uncompressed, with/without sound, etc.), purpose of use, etc. The curator will evaluate the user’s CineGrid membership status to see if he is in good standing with a signed usage agreement. Though presently these steps are handled manually, a fully automated workflow is planned in the near term and will be accomplished through a combination of web applications and CX middleware.

Following an initial request for content, the metadata management interface of CX will be searched to identify assets for distribution. The availability of the requested content will be returned from the search. A resource management module in the CX will identify the best resource to fetch the content, set the parameter and start the transfer using either TCP or UDP. All fetched files will have their checksums calculated and compared to the source automatically after the transfer. If the requester doesn’t have network access to a CineGrid node, an external hard drive will be used to distribute the content.

Once the workflow is finished, the curator and requester will receive confirmation emails from the system, and the CX curator will follow-up with the requester to confirm that the content is correctly received.

The design goals of the application workflows in the CX are to make it easier for applications running on top of the CX. While CX development to date has focused on implementing basic repository functions such as ingest, distribution, preservation and deletion, it is anticipated that more complex workflows can be implemented using the current platform with additional software efforts.

For example, a content creator (A) will create a clip that another CineGrid member (B) wants to view. However, suppose that B doesn’t have enough storage space to hold the uncompressed content and can only view compressed 4K clips, but A doesn’t have the equipment to compress the content. If both A and B are connected to the CX platform, an application workflow can resolve this problem by leveraging other available CineGrid resources. After A created the content, the content is ingested into the CX. Assuming user C has a codec that can compress the content, C can retrieve the con-
tent and compresses it and ingest it to the CX again. B can now stream the compressed content from the CX to his 4K projectors using a streaming protocol in the CX, viewing the content in real time.

This application example uses various components of the basic ingest, distribution workflows. More complicated workflows will clearly evolve as application requirements expanded and further functional requirements are explored. Demands faced by the digital motion picture industry will be a driver for further research and development in this area.

5.3.3 Workflow Description and Implementation

The workflows in the CX are described using iRODS rule description languages and are implemented as micro-service modules within iRODS using a high level C programming language. The iRODS rules are scripts that will be explained by the iRODS engine running as a server in the central server in the CX. Adding a new rule using existing micro-services will be immediately effective without recompiling or restarting any of the iRODS servers. Only administrators of the CX can add/remove/modify rules in the CX. If existing micro-services are not sufficient, users can also program new micro-services using iRODS micro-service APIs. This is also a privilege of the CX system administrators and will require recompiling and restart the CX system.

In general, the rules are declared in the singular iCAT server in the CX, and the execution of workflows can be either centralized or distributed. This depends on the APIs provided by the iRODS system and how the rules and micro-services are implemented by the users. In the current CX implementation, all rules are executed in one centralized place-the iCAT server in the CX. However, multiple rules can run in parallel. For example, the ingest flows can be invoked by multiple users at the same time, and the file transfers can be performed simultaneously with separate iRODS agents.

5.4 Experiments

The first phase of the CX is online now and is accessible to CineGrid members as a distributed storage platform, which manages all CineGrid assets and provides support
for distributed content access. Tens of terabytes of digital contents have been ingested through CX interfaces. The total size of the contents in the CX is about 50TB now, and is growing rapidly. In addition to its practical function, the CX is also used as a large-scale test bed for research purposes.

In this section, we will experiment and evaluate the CX in two major aspects: basic functions and workflow experiences.

5.4.1 CX Functionalities

CX is a distributed storage system, which can be used to put/get files. Figure 5.4a shows a scenario where the system administrator, node operators or the CX curator can use the iRODS command line interface to navigate current CX directories, put files into the CX for retrieval, etc. The figure also shows that the CX platform exists in parallel with a local file system, where each can freely accessed. The convenience provided by this approach is easier to use than a traditional FTP server.

Figure 5.4b is the browser interface, which can be accessed from any browser with Internet access. Files stored in the CX can be previewed (assuming there is a reduced-bitrate proxy or thumbnail) through this interface. For example, there are many versions with the same title, and it is often difficult for a system administrator to readily identify files. The browser interface can facilitate communication with the content creator or other systems administrators without the need of cumbersome downloading and converting files, emailing large attachments, or verbally describing the content in question.

Table 5.2: The CX file transfer performance

<table>
<thead>
<tr>
<th>Network Bandwidth</th>
<th>10Gbps Source</th>
<th>10Gbps Destination</th>
<th>10Gbps Source</th>
<th>10Gbps Destination</th>
<th>10Gbps Source</th>
<th>10Gbps Destination</th>
<th>10Gbps Source</th>
<th>10Gbps Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Tokyo, JP</td>
<td>San Diego, US</td>
<td>Amsterdam, NL</td>
<td></td>
<td>Tokyo, JP</td>
<td>San Diego, US</td>
<td>Amsterdam, NL</td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td>San Diego, US</td>
<td>Amsterdam, NL</td>
<td>Tokyo, JP</td>
<td></td>
<td>San Diego, US</td>
<td>Amsterdam, NL</td>
<td>Tokyo, JP</td>
<td></td>
</tr>
<tr>
<td>Read Speed</td>
<td>4.9Gbps</td>
<td>5.4Gbps</td>
<td>5.3Gbps</td>
<td></td>
<td>4.9Gbps</td>
<td>5.4Gbps</td>
<td>5.3Gbps</td>
<td></td>
</tr>
<tr>
<td>Write Speed</td>
<td>3.8Gbps</td>
<td>3.8Gbps</td>
<td>3.6Gbps</td>
<td></td>
<td>3.8Gbps</td>
<td>3.8Gbps</td>
<td>3.6Gbps</td>
<td></td>
</tr>
<tr>
<td>Disk-2-Disk TCP (1 stream)</td>
<td>160Mbps</td>
<td>150Mbps</td>
<td>150Mbps</td>
<td></td>
<td>160Mbps</td>
<td>150Mbps</td>
<td>150Mbps</td>
<td></td>
</tr>
<tr>
<td>Disk-2-Disk UDP (1 stream)</td>
<td>1.0Gbps</td>
<td>1.0Gbps</td>
<td>0.9Gbps</td>
<td></td>
<td>1.0Gbps</td>
<td>1.0Gbps</td>
<td>0.9Gbps</td>
<td></td>
</tr>
</tbody>
</table>
(a) The CX access commands interfaces through iRODS

(b) The CX access web interface (through iRODS) 4K images, for example a GLIF map, can be retrieved through the interface.

**Figure 5.4**: CX access interfaces

As a distributed platform, transfer speed between many sites is of critical importance. CX data transfer operations are done through its file transfer protocols, integrated
in iRODS. Initially, iRODS only supported the TCP protocol; UDP was added into iRODS as part of early CX development efforts to boost the performance. Better protocols to further utilize 10Gbps bandwidth connecting most CX nodes are being explored. Table 5.2 summarizes the current data transfer speed in the CX among the three major sites.

Table 5.2: Summary of current data transfer speed in the CX among the three major sites.

<table>
<thead>
<tr>
<th>Site</th>
<th>Current Data Transfer Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Diego</td>
<td>100 Mbytes/sec</td>
</tr>
<tr>
<td>Tokyo</td>
<td>80 Mbytes/sec</td>
</tr>
<tr>
<td>New York</td>
<td>120 Mbytes/sec</td>
</tr>
</tbody>
</table>

Figure 5.5: Parallel transfer throughput from San Diego to Tokyo (X-axis: number of parallel transfers; Y-axis: throughput in Mbytes/sec)

A 10Gbps interconnection is typical for the CX. However, this high bandwidth network seems over provisioned based on current available network protocols, because none of the current protocols in the CX can entirely fill the 10Gbps pipes. For TCP, due to the high Round Trip Time (RTT) value, even iRODS using 16 parallel TCP threads, the throughput is still only <2% of the capacity of the network bandwidth. Our UDP protocol is derived from single-threaded RBUDP, and can achieve around 10% of the maximum bandwidth. On the other hand, we can improve the performance of network bandwidth utilization by executing parallel tasks. We have experimented on the CX platform with parallel data transfer between the CX sites. Figure 5.5 shows one of them, which transfers a series of 1GB digital files from San Diego to Tokyo via the 10Gbps optical networks in the CX. The measurements are the overall throughput of all parallel transfers based on UDP and TCP respectively, and are plotted as we increase the number of parallel transfers. With parallel transfers, TCP throughput increases linearly and reaches a maximum of around 240Mbytes/sec with 14 parallel streams or more.
UDP reaches its maximum of 360Mbytes/sec with 3-4 parallel streams, and degrades significantly with more parallel streams, caused by higher UDP packet loss rate. The bottlenecks are protocols, disk I/O speed, memory copies, etc.

Therefore, in the CX, the way to really fill a 10Gbps pipe is to use a parallel data transfer protocol to transfer data from multiple nodes to multiple nodes. Some related work has been done on scalable parallel transfer protocols, like CSTP, Multi-Rail [VLTS08] and has gotten preliminary results, but they have not been integrated the protocol into the CX yet.

Nevertheless, even using the current CX implementations, transferring terabytes of data from Japan to USA only takes a few hours. In our past experience, the CX platform has been used by CineGrid members to move around large amount of critical data to meet urgent deadlines, which would not be possible in any other way. The relatively high file transfer speed also is a good foundation for digital contents workflows.

5.4.2 CX Policy Implementation and Evaluation

CX uses iRODS rules for the major part of CX workflows. A rule in iRODS defines a series of functions that will be activated at a certain point in time. All rules are listed in a few text files located at each storage server in the CX. The rule engine running at the same server will sequentially examine all the rules. Once a match is found, the rule engine will activate the rule. The structure of a rule is “ruleName | conditions | operations | recoveries”. When activated according to ruleName, the rule will first check conditions, and if “true”, it will execute the operations. The recoveries are used to handle exceptions during the execution of operations. The operations and recoveries of the rule are programmable using the microservice interface of iRODS. A workflow may consist of multiple rules, and each rule may consist of more than one operations/recoveries.

For example, the three rules associated with the CX ingest workflow are shown as below.

```
acPostProcForPut | $objPath like /home/user/dropBox/* | msiDataObjRepl($objPath, CineGrpResc, *st) $$
msiDataObjRepl($objPath, CineGrpResc, *st)$$ msiDataObjAutoMove($objPath,
```


The first rule “acPostProcForPut” is the post-process rule and will be activated upon the completion of ingesting files/directories. Once activated, it will first check the path of the file/directory, and if it is in the node operator’s dropBox, the rule will call a microservice “msiDataObjAutoMove”, written by us, to move the files into the repository. Following that we made two replicas to other sites in the resource group “CineGrpResc”, by calling another microservice “msiDataObjRepl”, which will automatically find a suitable available resource based on the resource management modules. The second rule disables the deletion permission from any user interface using the “msiDeleteDisallowed” microservice and ties to the “deletion workflow” which is not described here. The third rule invokes a delayed execution for doing content auditing in a two-month cycle and ties to the “preservation workflow” which is not described here. This microservice guarantees the integrity of the three replicas by recovering a failed replica or making one more replica to another available storage resource in the resource group. Lastly, it will mark the failed replica “obsolete” in the database.

Since the deployment of the CX workflows in September 2009, a few terabytes of digital content have been ingested through the defined workflows. Our experiences with the ingest workflow suggests checksums may be a potential bottleneck. When ingesting 50MB image files, the overhead of checksum is about 25%; but when ingesting 1GB files, the checksum overhead will be more than 50%. This is because the transfer protocol will be faster when the file size increases, while the checksum will be slower. This means that checksum overhead is substantial. It is expected that the ingest workflow can be revised to postpone the checksum operation to the auditing workflow, so that ingest workflow will finish much sooner.

For the distribution process, we have tested it by distributing several clips to a few CineGrid members. Groups at NPS, AMPAS, and SDSC are working on the metadata management (CollectiveAccess) and on the integration of CollectiveAccess
with iRODS. The full distribution workflow will be online when this is completed.

5.5 Conclusion And Future Work

After the first phase of design and implementation, CX has become an online distributed platform with three hierarchical layers. At its infrastructure layer, the physical resources have 250TB raw high-speed disk capacity at seven global major sites with 10Gbps interconnections. At the middleware layer, iRODS manages all the different types of storage resources, provides easy file access commands and programmable rules to support CX policies. At the application level, CX has implemented the first set of workflows for content ingestion, distribution, preservation and deletion. The structure of CX is completed and ready for further developments. The three first goals of CX have been accomplished: large amounts of digital data transfers are occurring regularly and frequently between CX sites using TCP or UDP protocols; a variety of research experiments are ongoing using the CX; and the first set of workflows are functioning well with limited administrative support.

In the next phase of CX refinement and development, we will address some more challenging functions. We will work more on resource management and scheduling; job scheduling for parallel access; streaming protocols to send digital content from CX directly to players. The current resource management scheme will be improved to find the right resource for the right application, automatic resource selection based on bandwidth/disk capacity/disk speed and geographical information. A distributed database is required for scalability and reliability reasons. More intelligent and reliable workflows, which are required to meet the request of digital applications, will be defined and evaluated. And further work towards integrating cluster resources, parallel transfer protocols, parallel storage access for content encoding/decoding as part of the CX will be pursued.

5.6 Acknowledgment

The work introduced in this chapter is a joint effort of The Academy of Motion Picture Arts and Sciences (AMPAS), San Diego Supercomputer Center, Calit2 at
UCSD, Naval Postgraduate School in Monterey, CA, EVL at the University of Illinois in Chicago, Keio University in Japan, CESNET in Czech Republic, UVA in the Netherlands, Ryerson University in Canada, University of Washington Research Channel, Pacific Interface in Oakland, CA, and many other CineGrid partners. The content of this paper has been submitted to the journal of Future Generation Computer Systems. Jurgen P. Schulze, Laurin Herr, Jeffrey D. Weekley, Bing Zhu, Natalie V. Osdol, Dana Plepys, Mike Wan are the co-authors. The dissertation author was the primary investigator and author of this material.
Chapter 6

Conclusion and Future Work

Future is mysterious. I personally have a strong belief that parallelism will be one of the major trends, especially when many techniques have advanced to a place close to their physical limits. For instance, when transferring data at a speed of megabyte per second, long distance seems not to be a factor that can significantly reduce the performance of a network application; but when transferring data at gigabyte per second, long distance becomes the major bottleneck that slows down the applications, while there is nothing we can do to increase the speed of light to make signals move faster in fibres.

In this thesis, we spend most of the pages on exploring the idea of using parallel data streams to transfer data between two distant locations. And we have proven that the Cross-Stream coding approach is indeed scalable to potentially unlimited number of parallel streams. We envision that Cross-Stream coding can be used for other purposes beyond the description in this thesis. For example, Cross-Stream coding may be suitable to construct a parallel storage systems which can reliable deliver data even some data on those disk drives are subject to loss. Compared to commonly used RAID systems, this approach can provide finer granularity of data recovery, while not requiring disks to be aggregated together.

We cannot predict the future, but maybe just because future is unpredictable, the exploration of the right directions becomes so interesting. That also concludes the meaning of spending years on a Ph.D program.
Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


