

TOWARDS DATA OPTIMIZATION IN STORAGEES AND NETWORKS

A DISSERTATION
IN
Telecommunications and Computer Networking
and
Computer Science

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
DAEHEE KIM

M. S., State University of New York, Binghamton, NY, USA, 2008
B. S., Pusan National University, Pusan, South Korea, 1995

Kansas City, Missouri
2015

© 2015
DAEHEE KIM
ALL RIGHTS RESERVED

TOWARDS DATA OPTIMIZATION IN STORAGE AND NETWORKS

Daehee Kim, Candidate for the Doctor of Philosophy Degree

University of Missouri–Kansas City, 2015

ABSTRACT

We are encountering an explosion of data volume, as a study estimates that data will amount to 40 zeta bytes by the end of 2020. This data explosion poses significant burden not only on data storage space but also access latency, manageability, and processing and network bandwidth. However, large portions of the huge data volume contain massive redundancies that are created by users, applications, systems, and communication models. Deduplication is a technique to reduce data volume by removing redundancies. Reliability will be even improved when data is replicated after deduplication.

Many deduplication studies such as storage data deduplication and network redundancy elimination have been proposed to reduce storage consumption and network bandwidth consumption. However, existing solutions are not efficient enough to optimize data delivery path from clients to servers through network. Hence we propose a holistic deduplication framework to optimize data in their path. Our deduplication framework consists of three components including data sources or clients, networks, and servers. The

client component removes local redundancies in clients, the network component removes redundant transfers coming from different clients, and the server component removes redundancies coming from different networks.

We designed and developed components for the proposed deduplication framework. For the server component, we developed the Hybrid Email Deduplication System that achieves a trade-off of space savings and overhead for email systems. For the client component, we developed the Structure Aware File and Email Deduplication for Cloud-based Storage Systems that is very fast as well as having good space savings by using structure-based granularity. For the network component, we developed a system called Software-defined Deduplication as a Network and Storage service that is in-network deduplication, and that chains storage data deduplication and network redundancy elimination functions by using Software Defined Network to achieve both storage space and network bandwidth savings with low processing time and memory size. We also discuss mobile deduplication for image and video files in mobile devices. Through system implementations and experiments, we show that the proposed framework effectively and efficiently optimizes data volume in a holistic manner encompassing the entire data path of clients, networks and storage servers.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled “Towards Data Optimization in Storages and Networks,” presented by Daehee Kim, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Sejun Song, Ph.D., Committee Chair
Department of Computer Science & Electrical Engineering

Baek-Young Choi, Ph.D. (Co-advisor)
Department of Computer Science & Electrical Engineering

Xiaojun Shen, Ph.D.
Department of Computer Science & Electrical Engineering

Ken Mitchell, Ph.D.
Department of Computer Science & Electrical Engineering

Lein Harn, Ph.D.
Department of Computer Science & Electrical Engineering

Yugyung Lee, Ph.D.
Department of Computer Science & Electrical Engineering

CONTENTS

ABSTRACT	iii
List of Figures	x
List of Tables	xiv
ACKNOWLEDGEMENTS	xv
Chapter	
1 Introduction	1
1.1 Redundancies	2
1.2 Existing Deduplication Solutions to Remove Redundancies	3
1.3 Issues of Existing Solutions	5
1.4 Objective of Ph.D. Study	6
1.5 Contributions	7
1.6 Organization	8
2 Deduplication Technology	10
2.1 Deduplication Classification	11
2.2 File-level Deduplication	13
2.3 Fixed-size Block Deduplication	14
2.4 Variable-size Block Deduplication	16
2.5 Comparison of Deduplications by Method per Granularity	17
2.6 Bloom Filter	18

2.7	Server Based Deduplication	21
2.8	Client Based Deduplication	22
2.9	End-to-end Redundancy Elimination	23
2.10	Network-wide Redundancy Elimination	24
2.11	Inline deduplication	28
2.12	Offline deduplication	29
3	Existing Deduplication Approaches	30
3.1	File-level Deduplication	30
3.2	Fixed-size Block Deduplication	32
3.3	Variable-size Block & Server-based Deduplication	36
3.4	Hybrid Deduplication	37
3.5	Object-level Deduplication	38
3.6	Client-based Deduplication & End-to-end Redundancy Elimination	39
3.7	Network-wide Redundancy Elimination	41
3.8	Inline & Offline Deduplication	43
4	HEDS: Hybrid Email Deduplication System	45
4.1	Large Redundancies in Emails	45
4.2	Hybrid System Design	47
4.3	EDMilter	48
4.4	Metadata Server	49
4.5	Bloom Filter	50
4.6	Chunk Index Cache	51

4.7	Storage Server	51
4.8	EDA (Email Deduplication Algorithm)	51
4.9	Experiment Setup	55
4.10	Deduplication performance	57
4.11	Memory overhead	62
4.12	CPU overhead	64
4.13	Summary	65
5	SAFE: Structure-Aware File and Email Deduplication for Cloud-based Storage Systems	66
5.1	Large Redundancies in Cloud Storage Systems	67
5.2	SAFE Modules	68
5.3	Email Parser	69
5.4	File Parser	71
5.5	Object-Level Deduplication and Store Manager	77
5.6	SAFE in Dropbox	77
5.7	Metrics and Setup	81
5.8	Storage and Data Traffic Reduction Performance	84
5.9	Memory and CPU Overhead	87
5.10	Summary	89
6	SoftDance: Software-defined Deduplication as a Network and Storage Service	91
6.1	Large Redundancies in Network	92
6.2	Software Defined Network	94

6.3	Control and Data Flow	95
6.4	Encoding Algorithms in Middlebox (SDMB)	98
6.5	Index Distribution Algorithms	100
6.6	Implementation: REST, JSON, Middlebox	107
6.7	Experiment and Emulation Setup	108
6.8	Storage Space and Network Bandwidth Saving	112
6.9	CPU and Memory Overhead	114
6.10	Performance and Overhead per Topology	115
6.11	SoftDance vs Combined Existing Deduplication Techniques	119
6.12	Summary	121
7	Mobile De-duplication	122
7.1	Large Redundancies in Mobile Devices	122
7.2	Approaches and Observations	123
7.3	JPEG and MPEG4	124
7.4	Throughput and Running Time of Encryption Algorithm	124
7.5	Summary	129
8	Conclusions	130
	REFERENCE LIST	132
	VITA	140

List of Figures

Figure		Page
1	Data explosion: (Source - IDC's Digital Universe Study, sponsored by EMC, December 2012)	1
2	Redundancies	2
3	Existing solutions to remove redundancies	4
4	Deduplication framework	7
5	Components developed for deduplication framework	7
6	File-level deduplication	13
7	Fixed-size block deduplication	15
8	Variable-size block deduplication	16
9	Comparisons of deduplications	18
10	How bloom filter works	19
11	Server based deduplication	21
12	Client based deduplication: c1 and c2 are chunks. h(c1) and h(c2) are hash keys (indexes) of chunks.	22
13	End-to-end redundancy elimination: c1 and c2 are chunks. h(c1) and h(c2) are hash keys (indexes) of chunks.	24
14	Network-wide redundancy elimination	25
15	Network-wide redundancy elimination: issue	25

16	Inline deduplication	27
17	Offline deduplication	29
18	A study of practical deduplication: evaluation setup	31
19	Venti tree structure of data blocks [63]	33
20	Dropbox internal mechanism	35
21	Sparse indexing: deduplicaiton process [44]	38
22	Low bandwidth file system (LBFS) [54]	40
23	Redundant traffic elimination with packet caches on routers	42
24	Proposed hybrid email deduplication system (HEDS)	47
25	EDMilter	49
26	Block deduplication at EDA	54
27	File-level deduplication at EDA	54
28	Distribution of the Enron corporate email sizes	56
29	Distribution of the gmail personal email sizes	57
30	Reduced storage (Enron - corporate dataset)	59
31	Reduced storage (Gmail - single user dataset)	60
32	Chunk index overhead	63
33	Relative CPU overhead	65
34	SAFE deduplication architecture	68
35	Email parser	70
36	Structure of an email	70
37	Physical file format	73

38	File parser	74
39	Logical structure of MS office document file	76
40	Dropbox internal mechanism	79
41	SAFE integration with Dropbox	80
42	Distribution of the file sizes in the email dataset	83
43	Percentage of the structured files in the email datasets	83
44	Deduplication ratio	85
45	Data traffic incurred (MB)	87
46	Relative processing time overhead compared to file-level deduplication	88
47	Relative index overhead compared to file-level deduplication	88
48	SoftDance architecture	93
49	Software defined network	95
50	SoftDance control and data flows	96
51	SoftDance forwarding table example	98
52	A network topology with three routes	102
53	JSON format example: response of hash range URI	108
54	Experiment topology	109
55	Emulation topology	110
56	Comparison of performance	112
57	Comparison of overhead	115
58	Memory size among SoftDance approaches: per-node	115
59	Performance per topology	117

60	Overhead per topology	118
61	Performance of combined approaches	120
62	Overhead of combined approaches	121
63	Throughput of encryption algorithms per file type	125
64	Running time of encryption algorithms per file type	126
65	Throughput of encryption algorithms per file size	128
66	Running time of encryption algorithms per file size	128

List of Tables

Tables		Page
1	Deduplication classification	11
2	Datasets	55
3	Locality of attachments	61
4	Used datasets	82
5	SD-uniform hash ranges and generated index size	103
6	SD-merge hash ranges and generated index size	104
7	REST API URIs	107

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisors including Dr. Sejun Song and Dr. Baek-Young Choi for their great advice and guidance for my Ph.D. study. Dr. Song enabled me to develop various multicasting projects for wireless sensor network with his insightful thoughts and helped me begin my current dissertation topic, deduplication. Dr. Choi's great guide for my study at the University of Missouri-Kansas City enabled me to learn research skills including writing, discovering ideas, building up projects, and ultimately making a paper throughout all research.

I would like to thank my committee members: Dr. Xiaojun Shen, Dr. Ken Mitchell, Dr. Lein Harn, and Dr. Yugyung Lee for all their help for my dissertation and insightful comments. Thanks to these comments, this dissertation was improved significantly.

I also sincerely thank Professor Judy Mullins. Thanks to her kind help and encouragement during my assistance as a graduate assistant for her data structure and algorithm course, I had great opportunities to experience Big Data using Hadoop and Amazon Cloud and IBM Cloud. Making the first Hadoop tutorial in UMKC with her was a great joyful moment. Throughout two and a half year of assistance to her, I learned great teaching skills from her.

I appreciate both my parents and my wife's parents, ByungJun Kim and Okyeun Ahn, HyungKi Shin and MyungJa Song. During my entire Ph.D. study, they always have been on my side to encourage and help me and my family. My beloved daughters Sarah

and Daisy have grown taller since we came here and their continuous smiles at me have been one of the greatest weapons and ways that I could continue my study without taking a break. I thank my lovely wife, Jongsoon Shin. Without her help and support, I could not achieve this amazing goal.

I thank God, Jesus who always lead and guide my way, and know better than me about my way even though I cannot see what life holds in store for me.

CHAPTER 1

INTRODUCTION

We live in the era of data explosion. Based on the IDC's digital universe study [34] as shown in Figure 1, data volume will increase by 50 times higher at the end of 2020 than it is in 2010, which amounts to 40 zetabytes (40 million petabytes - more than 5,200 gigabytes for every person). This huge increase of data volume has a critical impact on the overhead costs of computation, storage, and network.

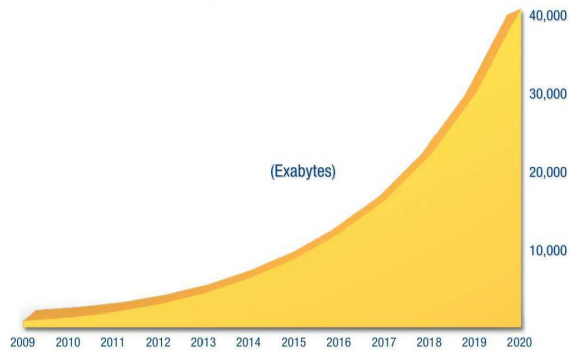


Figure 1: Data explosion: (Source - IDC's Digital Universe Study, sponsored by EMC, December 2012)

Interestingly, massive portions of this enormous data are derived from redundancies in storages and networks. A study [46] shows there is a redundancy of 70% in datasets collected from file systems of almost one thousand computers in an enterprise. Another study [71] finds that 30% of incoming traffic and 60% of outgoing traffic are redundant based on packet traces on a corporate research environment with 3000 users and web servers.

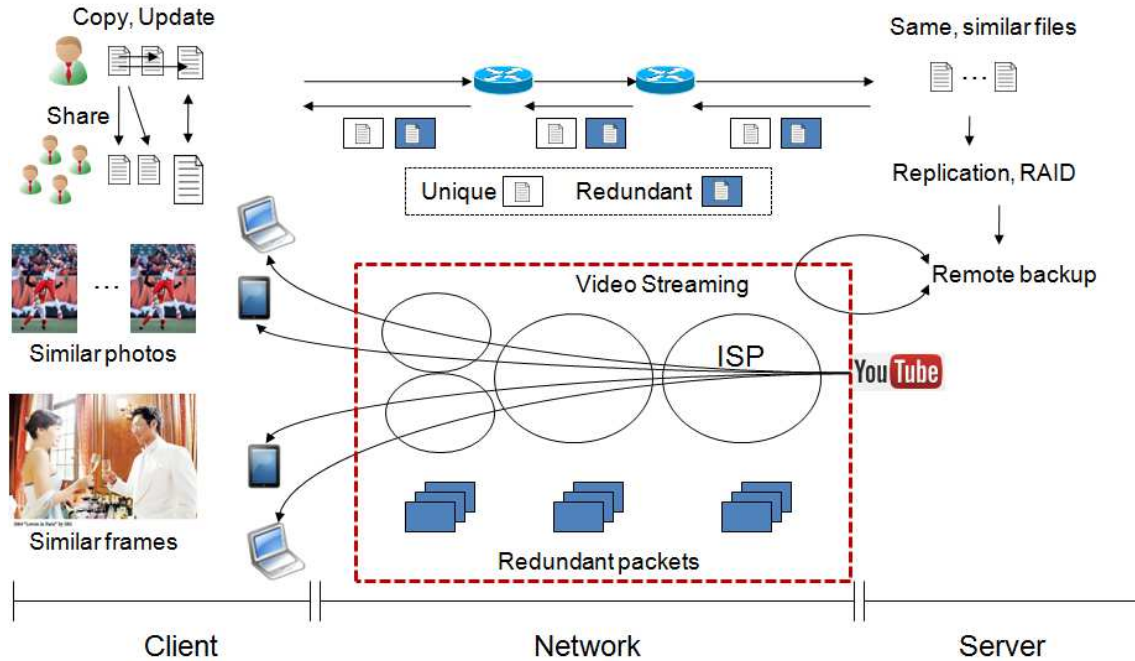


Figure 2: Redundancies

1.1 Redundancies

Redundancies are produced in clients, servers, and networks in various manners as shown in Figure 2. Redundancies increase on the client side. A user copies a file with a different file name and creates similar files with small updates. These redundancies further increase when users copy redundant files back and forth among people within an organization. Another type of redundancy is generated by applications. For example, currently there is a popular trend to take pictures of moving objects, called burst shooting mode. In this mode, we can take 30 pictures within a second, and choose good pictures or remove bad pictures. However, this application can produce large redundancies among similar pictures. Another type of redundancy occurs from similar frames in video files.

A video file consists of many frames. In scenes where actors keep talking with the same background, large portions of the background become redundancies.

Redundancies also occur on the network side. When a user requests a file for the first time, a unique transfer occurs, which does not produce redundant transfers in a network. However, when a user requests the same file again, redundant transfer occurs. Redundancies are also generated by data dissemination such as video streaming. For example, when different clients receive a streaming file from Youtube, redundant packets must traverse through multiple Internet Service Providers (ISPs).

On the server side, redundancies are greatly expanded when people in the same organization upload the same (or similar) files. The redundancies are accelerated by replication, Redundant Array of Inexpensive Disks (RAID), and remote backup for reliability.

Then, what problems arise from these redundancies? From the client and server sides, storage consumption increases. On the network side, network bandwidth consumption increases. For clients, latency increases because users keep downloading the same files from distant source servers each time. We find that redundancies significantly impact storages and networks. Then, the next question is: what solutions exist to remove (or reduce) these redundancies.

1.2 Existing Deduplication Solutions to Remove Redundancies

As shown in Figure 3, there are three type of studies throughout storages and networks. The first type is storage data deduplication that aims to save storage spaces. In

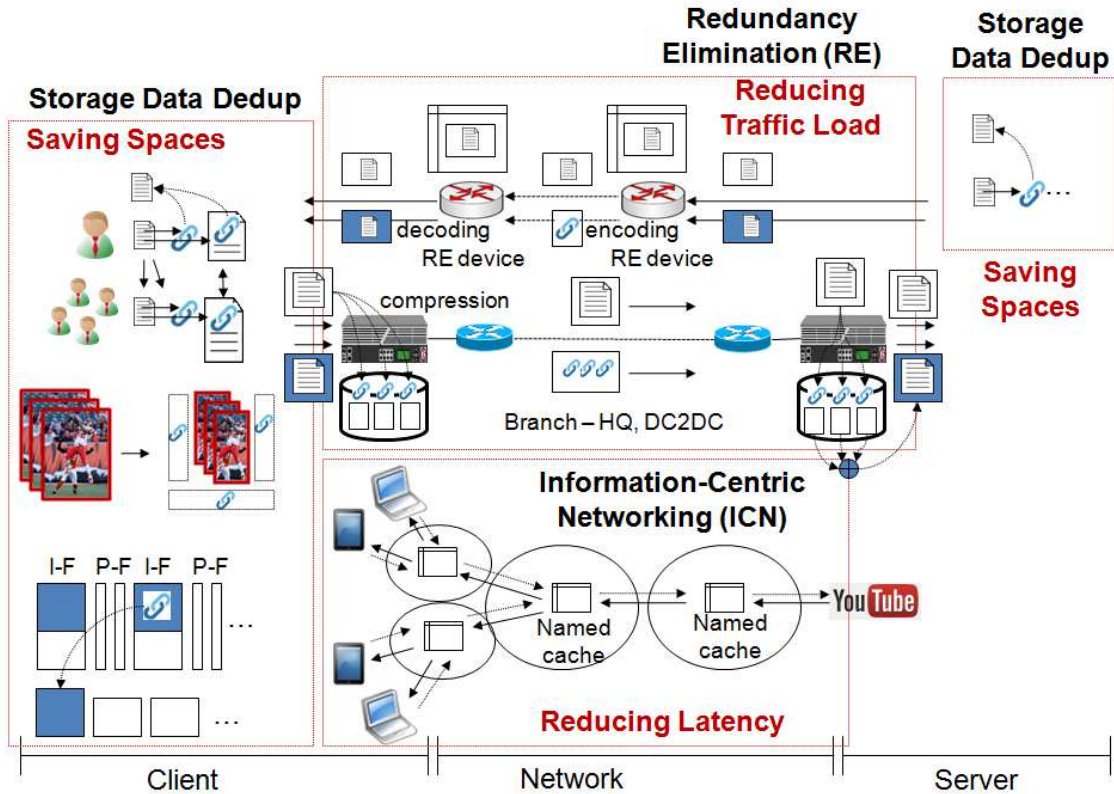


Figure 3: Existing solutions to remove redundancies

this approach, only a unique file or chunk is saved, but redundant data are replaced by indexes. Likewise, an image is decomposed into multiple chunks, and redundant chunks are replaced by indexes. A video file consists of I-frame that has the image itself and P-frame that has the delta information between images in I-frames. In a video file where the backgrounds are same, I-frames have large redundancies that are replaced by indexes. Servers deduplicate redundancies coming from clients by using storage data deduplication.

The second type is Redundancy Elimination (RE). This aims to reduce traffic loads in networks. The typical example is the Wide Area Network (WAN) optimizer that removes redundant network transfers between branches (or a branch) to a head quarter and

a data center to a different data center. WAN optimizer works like this. Suppose a user sends a file to a remote server. When the file is passing, the WAN optimizer splits the file into chunks and saves the chunks and corresponding indexes. The file is compressed and delivered to the WAN optimizer at other side where the file is again split into chunks which are saved along with indexes. Next time when the same file passes, the WAN optimizer replaces the file with small sized indexes. The WAN optimizer at the other side reassembles the file with previously saved chunks based on indexes on a packet. The other example is network-wide redundancy elimination, network-wide RE where a router (or switch) is specially named as an RE device. In this approach, for a unique transfer, RE devices save unique packets. When transfers become redundant, an RE device replaces redundant payload within a packet with an index (called encoding), and reconstructs the encode packet (called decoding).

The third type is Information Centric Networking (ICN) that aims to reduce latency. In ICN, any router can cache data packets passing by. Thus, when a client requests data, any router with the proper cache can send the requested data.

1.3 Issues of Existing Solutions

Problems exist within these current solutions. First, storage data deduplication has considerable computation and memory overhead in clients or servers. Many studies are focused on the trade-off between space savings and overhead based on granularity. Using small sized granularity like 4 KB can find more redundancies than using large sized granularity such as a file, but it suffers from large processing time and index overhead.

Second, redundancy elimination (RE) has resource-intensive operations such as fingerprinting, encoding, and decoding at routers. Also a representative RE study suggests a control module that involves traffic matrix, routing policies, and resource configurations, but there are not many details and some of them are based on assumptions. Thus, we need to have an efficient way to adapt RE devices with dynamic changes. Third, information centric networking (ICN) uses name-based forwarding tables that grow much faster than IP forwarding tables. Thus, large table lookup times and scalability issues arise.

1.4 Objective of Ph.D. Study

To remove (or reduce) issues of existing solutions, the objective of my Ph.D. study is to develop a deduplication framework that optimizes data from clients to servers through networks. The framework consists of three components that have different levels of removing redundancies as shown in Figure 4.

The client component removes local redundancies with a client and is basically comprised of functions to decompose and reconstruct a file. These components should be fast and have low overhead considering the low-capacity of most clients. The network component removes redundant transfers coming from different clients. In this component, RE devices intercept data packets and eliminate redundant data. RE devices are dynamically controlled by Software Defined Network (SDN) controllers. This component should be fast analyzing large amount of packets and should be scalable to a large number of RE devices. Last, server component removes redundancies coming from different networks. This component should satisfy high space savings. Thus, fine-grained deduplication and

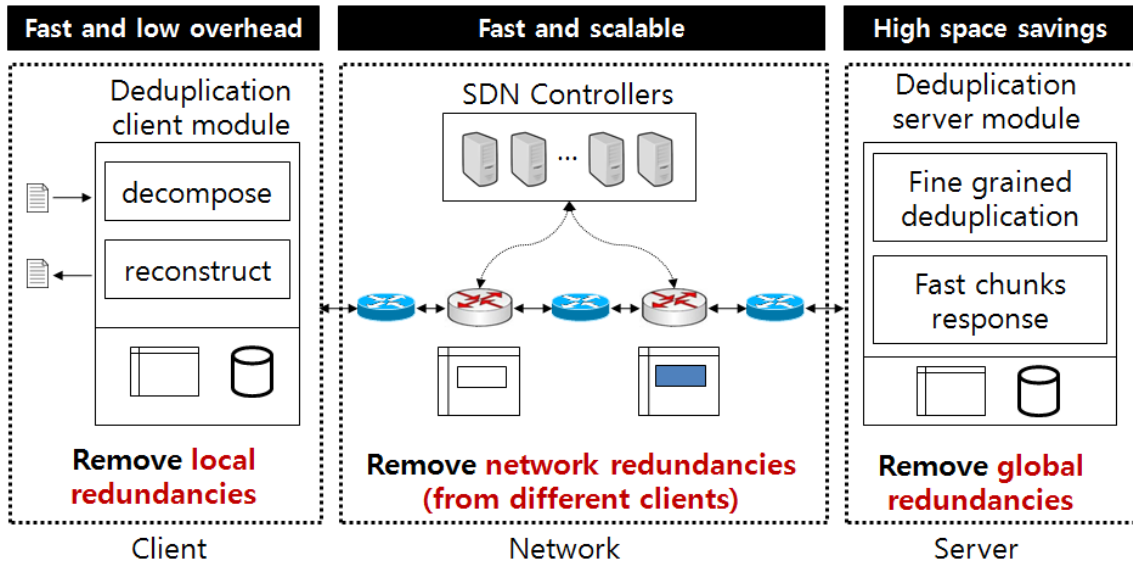


Figure 4: Deduplication framework

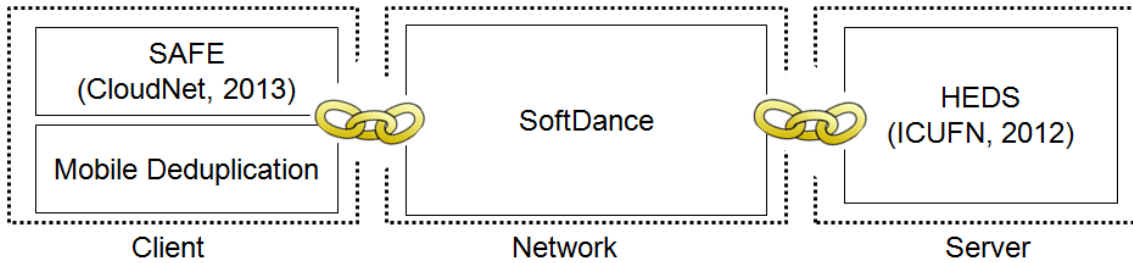


Figure 5: Components developed for deduplication framework

fast response are fundamental functions.

1.5 Contributions

We propose a deduplication framework using components we designed and developed for the proposed deduplication framework as shown in Figure 5. For the server component, we developed Hybrid Email Deduplication System, HEDS, that achieves a balanced trade-off of space savings and overhead for email systems. This work has

been published in IEEE International Conference on Ubiquitous and Future Networks (ICUFN), 2012. For the client component, we developed Structure-Aware File and Email Deduplication for Cloud-based Storage Systems, SAFE that is fast and has high storage space savings by using structure-based granularity. This work has been published in IEEE 2nd International Conference on Cloud Networking(CloudNet), 2013. For the network component, we developed Software-Defined Deduplication as a Network and Storage Service, or SoftDance. SoftDance is an in-network deduplication that chains storage data deduplication and redundancy elimination functions by using Software Defined Network (SDN), and achieves both storage space and network bandwidth savings with low processing time and memory overhead. This work will be submitted to a premier conference and we applied for invention disclosure. For the extension of the client component, we are also working on mobile deduplication that removes redundancies of popular files like images and video files in mobile devices. Part of this work has been submitted to a book chapter.

1.6 Organization

This dissertation follows the order of components that we developed for the deduplication framework. We give background information about how deduplication works in Section 2, and introduce existing deduplication studies in Section 3. After that, we elaborate on each component for the deduplication framework one by one. In Sections 4 and 5, we present a server component and a client component: Hybrid Email Deduplication Systems (HEDS) and Structure-Aware File and Email Deduplication for Cloud-based Storage

Systems (SAFE) respectively. In Section 6, we elaborate how deduplication can be used for networks and storages to reduce data volumes by using Software-defined Deduplication as a Network and Storage Service, or SoftDance. We show our on-going project, mobile deduplication, in Section 7. Section 8 concludes this dissertation.

CHAPTER 2

DEDUPLICATION TECHNOLOGY

Though various deduplication techniques have been proposed and used, there has been no one-best-fit solution to handle all types of redundancies. Considering performance and overhead, each deduplication technique has been developed with different designs' considering characteristics of datasets, system capacity, and deduplication time. For example, if datasets to be handled have many duplicate files, deduplication can compare files themselves without looking inside the file content for faster running time. However, if datasets have similar files rather than identical files, deduplication should look inside the file content to check what parts of the file content are the same as previously saved data for better storage space savings. Also deduplication should consider different designs up to the capacity of the system. High capacity servers can handle considerable overhead for deduplication, but low capacity clients should have lightweight deduplication designs for fast performance. There have been studies to reduce redundancies at routers (or switches) within a network. This approach requires the fast processing of data packets at routers, which is of great necessity for Internet Service Providers (ISPs). Meanwhile, if a system should remove redundancies directly in a write path within a confined storage space, it is better to discover redundant data before storing. On the other hand, if a system has residual (or idle) time or space enough to store data temporarily, deduplication can run after data are placed into temporary storages.

Table 1: Deduplication classification

Methods per granularity	Place	Time
File-level deduplication Fix-size block deduplication Variable-size block deduplication	Server-based deduplication Client-based deduplication Redundancy elimination(RE) (End-to-end RE, Network-wide RE)	Inline deduplication Offline deduplication

In this chapter, we classify existing deduplication techniques based on deduplication methods per granularity to be used, deduplication place, and deduplication time. Then, we describe how each deduplication technique works along with existing approaches in brief. We elaborate detail of existing commercial and academical existing deduplication approaches in next chapter.

2.1 Deduplication Classification

Deduplication can be divided based on methods per granularity (the unit of compared data), deduplication place, and deduplication time, as shown in Table 1. The main components of these three classification criteria are chunking, hashing, and indexing. Chunking is a process that generates the unit of compared data, called a chunk. To compare duplicate chunks, hash keys of chunks are computed and compared, and a hash key is saved as an index for future comparison with other chunks.

For methods, deduplication is based on different granularity. The unit of compared data can be file-level or sub-file level which are further divided into fixed-size block, variable-size chunk, packet payload, or byte streams in a packet payload. The smaller granularity is used, the more indexes are created, but the more redundant data are detected

and removed.

For deduplication place, deduplication is divided into server-based and client-based deduplication for end-to-end systems. Server-based deduplication traditionally runs on high capacity servers whereas client-based deduplication runs on clients that have normally limited capacity. Deduplication can occur on the network side, called Redundancy elimination (RE). The main goal of *redundancy elimination* techniques is to save bandwidth and reduce latency by reducing repeating transfers through network links. Redundancy elimination is further divided into end-to-end RE where deduplication runs at endpoints in network and network-wide RE (or in-network deduplication) where deduplication runs at routers in network.

For deduplication time, deduplication is divided into Inline and Offline deduplication. Inline deduplication runs deduplication before data are stored into disks whereas Offline deduplication runs deduplication after data are stored. Thus, Inline deduplication does not require extra storage space but incurs latency overhead within a write path. Oppositely, Offline deduplication does not have latency overhead but involves extra storage space and more disk bandwidth because data saved in temporary storage are loaded for deduplication and deduplicated chunks are saved again into more permanent storage. Inline deduplication mainly focuses on latency-sensitive primary workload, whereas Offline deduplication concentrates on throughput-sensitive secondary workload. Thus, Inline deduplication studies tend to trade-off storage space savings for fast running time.

We elaborate classified deduplication techniques in detail one by one hereafter, in the order of methods, place, and time. Please note that a deduplication technique can

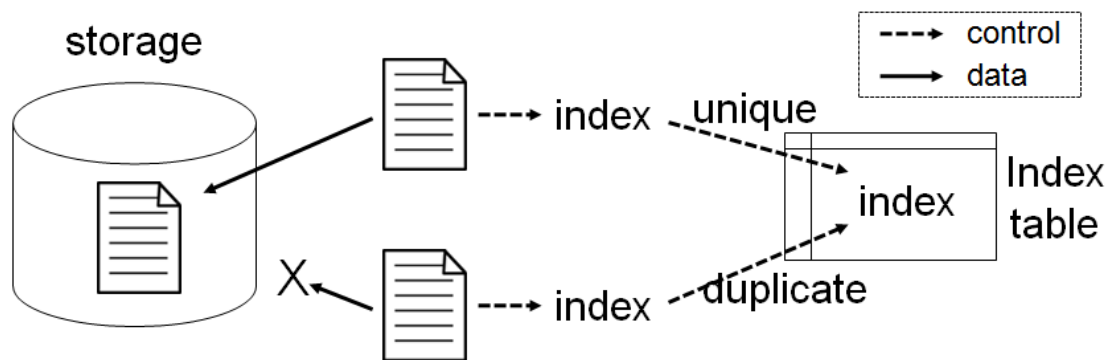


Figure 6: File-level deduplication

belong to multiple categories like a combination of variable-size block deduplication, server-based deduplication, and inline deduplication.

2.2 File-level Deduplication

File-level deduplication uses file-level granularity which is the most coarse-grained granularity. File-level deduplication compares entire files based on a hash value of a file like SHA1 [56] to avoid saving the same files. For example, as shown in Figure 6, suppose we have two identical files. When we save the first file, deduplication computes an index which is a hash value by one-way hash function. If the index is not found in index table, the file is unique. In this case, the index and the file are saved into index table and storage respectively. For the second file, the index of the file is found in the index table, so the corresponding file is not saved.

File-level deduplication has been used for removing redundancies of identical files in storage, email systems, cloud-based storage systems. For storage, EMC's Centera [23]

uses file-level deduplication to reduce redundancies in storage. For email systems, Microsoft Exchange 2003 [47] and 2007 [48] use file-level deduplication, called Single Instance Store (SIS) [7]. An email with multiple recipients is copied into multiple mailboxes, resulting in having multiple copies of the email. In this case, SIS saves only one copy of an email in recipient’s mailbox and saves only the pointers of the email in other recipients’ mailboxes without storing the email redundantly in individual recipients’ mailboxes. Many cloud-based storage services such as JustCloud [39], and Mozy [52] also use file-level deduplication. There has been a study [46] on a corporate users’ file systems where simple file-level deduplication can achieve three quarters of the space savings of aggressive expensive block deduplications (which we will discuss at the next two sections) at a lower cost in performance and complexity.

2.3 Fixed-size Block Deduplication

File-level deduplication can find redundancies of identical files, but cannot find redundancies within similar files. To find redundancies in similar files, fixed-size block deduplication has been proposed and uses fixed-size block as its granularity. However, fixed-size block deduplication has an issue finding matching contents in similar files when content in the beginning of files is changed. For example as shown in Figure 7, suppose deduplication uses 15 byte fixed-size block as granularity. When we save an original file *File1*, deduplication splits a file into 15 byte fixed-size blocks. Likewise, when we save an updated file *File2* where we add small text “welcome” in the beginning of the original file, deduplication again splits a file into fixed-size blocks. However, blocks split from the

e.g. granularity : 15 byte fixed size

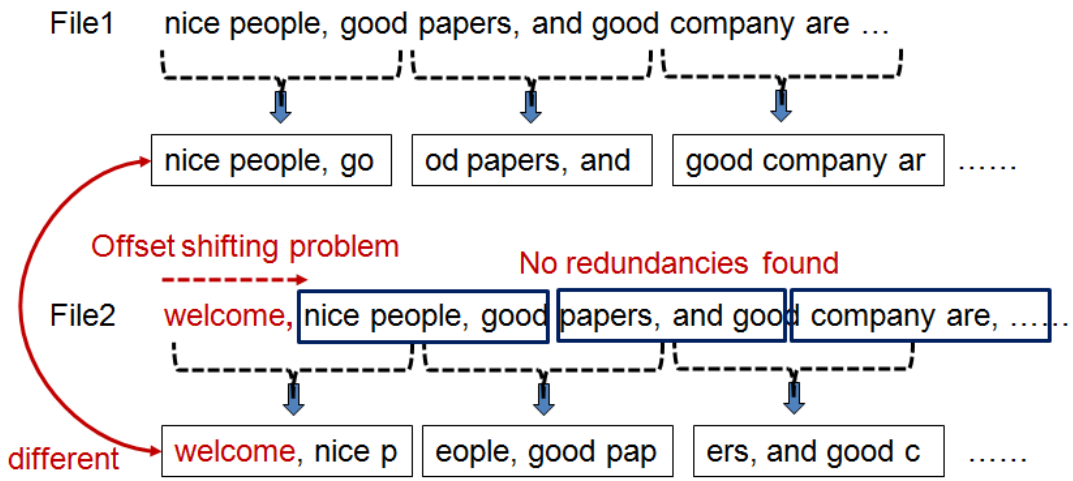


Figure 7: Fixed-size block deduplication

updated second file are totally different from blocks split from the original first file. This is because contents are shifted in a file, and this is called *offset-shifting problem*.

Fixed-size block deduplication has been used for archival storages like Venti [63]. Venti uses fixed-size block as granularity and compares SHA1 hash keys of blocks with previously saved hash keys following on-disk index hierarchy. A popular cloud storage system, Dropbox [14] uses a very large fixed-size (4 MB) block deduplication. Dropbox reduces network redundant traffic and redundant savings in server by communication with indexes between clients and servers before sending data. Detailed information how Dropbox works is explained in Chapter 5.

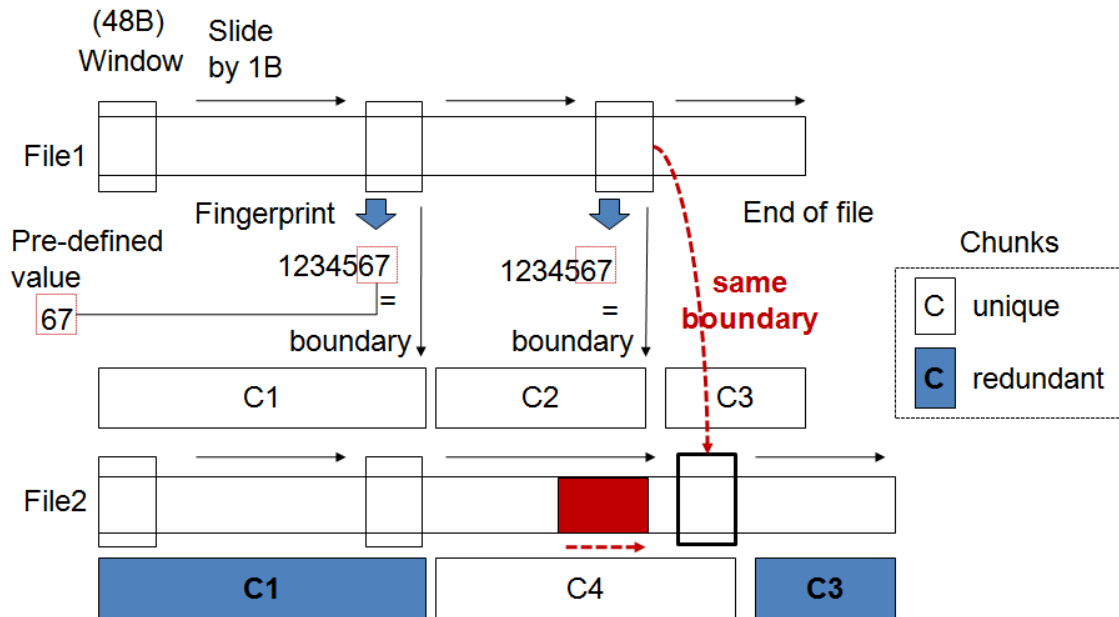


Figure 8: Variable-size block deduplication

2.4 Variable-size Block Deduplication

Variable-size block deduplication has been proposed to solve the *offset-shifting problem* of fixed-size block deduplication. Variable-size block deduplication relies on contents rather than fixed-offset. Figure 8 illustrates how variable-size block deduplication works. Suppose we have two files. *File1* is an original file and *File2* is an updated file where we add a small text in the middle of a file. When we save the *File1*, deduplication slides a small sized window from the beginning of the file. While the window is sliding byte by byte, a fingerprint [64] of each window is computed and the lowest two digits are compared to a pre-defined value. If they are the same, the window is set to a chunk boundary. Then, the content from the previous chunk boundary to the current chunk boundary are determined as a chunk. The window keeps sliding and finding chunk boundaries in

the same manner. As a result, three unique chunks ($C1$, $C2$, and $C3$) and corresponding indexes are saved. When we save the updated second file, deduplication again slides a window and finds chunks. $C4$ is found to be unique, and $C1$ and $C3$ are found to be redundant. Here, we see chunk boundaries are maintained though contents are shifted in a file. Thus, content-based variable-size block deduplication can find more redundancies than offset-based fixed-size block deduplication.

Since variable-size block deduplication provides fine granularity chunking techniques to achieve high storage space savings, it has been used for backup [12] [16] [29] [44] [76] [78] or file systems [8] [70]. However, to speed up processing time by reducing number of disk accesses, this approach like Data Domain File System (DDFS) [78] exploits efficient caching schemes like bloom filter and chunk index cache, and locality-based disk layout.

2.5 Comparison of Deduplications by Method per Granularity

Overall, as shown in Figure 9, the deduplication ratio that indicates how many redundancies are removed, variable-size block deduplication is much better than others. For processing time, variable-size block deduplication is the worst due to expensive chunking. For index overhead, fixed-size and variable-size block deduplication is much worse than file-level deduplication, and index overhead of fixed-size and variable-size block deduplication changes depending on block or chunk size. Thus, variable-size block deduplication is good for deduplication of updated files or server-based deduplication because high capacity servers can handle excessive processing time and index overhead. On the other




	Good for client-based Or copied files		Good for server-based Or updated files
 Deduplication ratio	File-level	< Fixed size << better	Variable size
 Processing time	File-level	< Fixed size <<<< worse	Variable size
 Index overhead	File-level	<< Fixed size \cong worse	Variable size

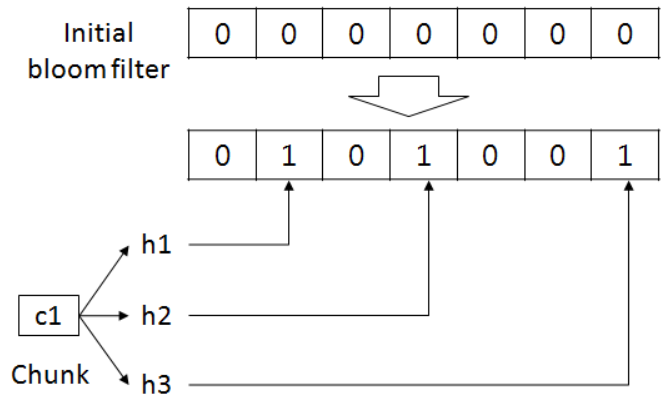
Figure 9: Comparisons of deduplications

hand, file-level deduplication is good for deduplication of the copied files or client-based deduplication considering low capacity clients.

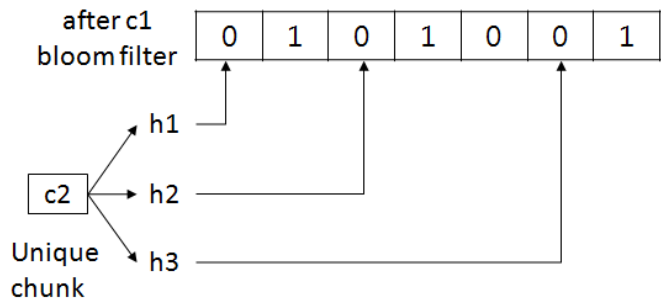
2.6 Bloom Filter

Deduplication aims to find as many redundancies as possible while maintaining processing time. To reduce processing time, one typical technique is to check indexes of data in memory before accessing disks. If the indexes of data are same, deduplication does not access the disk where indexes are stored, which reduces processing time.

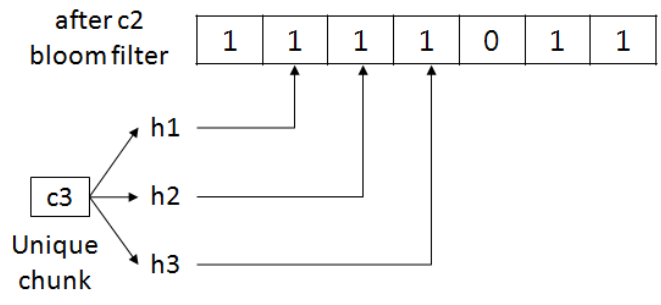
A Bloom filter is used to see if duplicate chunks of a data exist in storage. The Bloom filter is a bit array of m bits, that are set to 0 initially. Given a set U , each element u ($u \in U$) of the set is hashed using k hash functions h_1, \dots, h_k . Each hash function $h_i(u)$ returns an array index in the bit array, that ranges from 0 to $m - 1$. Subsequently, a bit



(a) bloom filter after c1 chunk is saved



(b) bloom filter when c2, a unique chunk is compared



(c) bloom filter when c3, a unique chunk is compared (false positive). A unique chunk is found to be redundant

Figure 10: How bloom filter works

of the index is set to 1. If the bit of the index was set to 1, it stays 1. This Bloom filter is used to check if an element was already saved into a set. When an element attempts to be added into the set, if one of the bits corresponding to the return values of hash functions h_1, \dots, h_k is 0, the element is considered as a new one in the set. If bits corresponding to return values of hash functions are all 1, the element is considered to exist in the set.

Let us explain how bloom filter works in an example. As shown in Figure 10(a), the bloom filter initially have all 0 bits. When a chunk c_1 is saved, the array indexes of bloom filter are computed by using three different hash functions (h_1 , h_2 , and h_3). Here, h_1 , h_2 , and h_3 functions return 2nd, 4th, and 7th index respectively. Subsequently indexes of bloom filter are set to 1. Suppose the same chunk c_1 is saved again. The chunk is found to be redundant because all three indexes by hash functions are already set to 1. As shown in Figure 10(b), when a unique chunk (c_2) is saved, indexes by three hash functions are computed again. Now, the elements of the three indexes are all 0. Thus, a chunk c_2 is determined to be unique. However, in Figure 10(c), bloom filter can have false positive; that is, bloom filter says that a chunk is redundant but the chunk is unique. The array indexes for c_3 are 2nd, 3rd, and 4th, which were set by other chunks. In this case, we will lose a unique chunk without saving it. Thus, bloom filter guarantees that a chunk is unique with any of one 0 index, but it does not guarantee that a chunk is redundant with all three 1 indexes. Thus, in this case, chunk index cache should be checked after bloom filter.

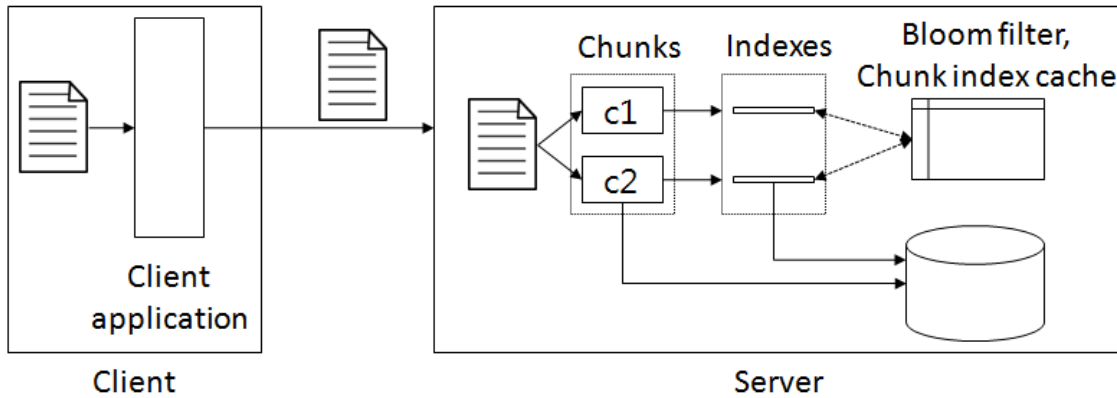


Figure 11: Server based deduplication

2.7 Server Based Deduplication

Server based deduplication has emerged as a disk-based substitute of tape storage, and backs up large size data at fast speeds using high performance and dedicated backup systems. There are many commercial products [24] [58] [74].

In this approach, clients send backup data to servers where data are de-duplicated. Clients have light-weight backup by application through which data are sent to servers, avoiding large CPU computation and memory overhead of sources for backup purposes. Figure 11 shows how server-based deduplication works. A file is transferred to a server through a client application. In the server, the file is separated to chunks typically using variable-size block deduplication. Indexes of chunks are computed and compared with indexes previously saved using a bloom filter or a chunk index cache. Suppose a chunk $c1$ is redundant and a chunk $c2$ is unique in this example. Then, a chunk $c2$ and its corresponding index are saved into storage.

Server based deduplication finds significant redundancies, but incurs excessive

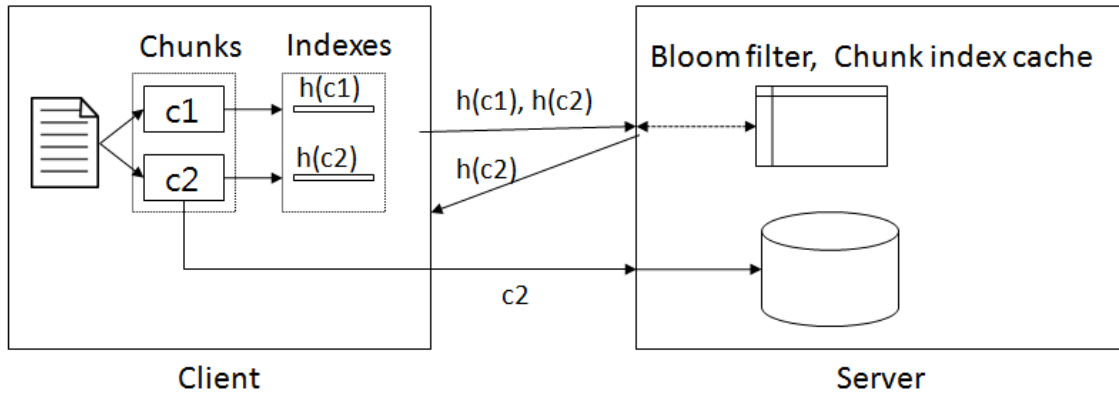


Figure 12: Client based deduplication: $c1$ and $c2$ are chunks. $h(c1)$ and $h(c2)$ are hash keys (indexes) of chunks.

redundant data traffic because duplicate data are delivered to servers to be deduplicated. What is worse, servers have large CPU computation and memory overhead for chunking and indexing of all backup data. To handle backup quickly with this overhead within a limited backup window, efficient in-memory and on-disk layout are required such as Data Domain File System (DDFS) [78].

2.8 Client Based Deduplication

In *client based deduplication*, clients can keep indexes of deduplicated data or have a backup agent to check indexes that exist in servers. In either case, clients check uniqueness of data in local indexes or in remote indexes through backup agent. Only unique data are then delivered to servers. Client based deduplication [22] [75] removes excessive redundant network traffic by performing deduplication at the client before data is transmitted. However, clients incur CPU computation and memory overhead for backup.

Pure client based deduplication, where a client removes redundant data before

sending data to a server, does not collaborate with a server (or servers); redundant data among clients are transferred to a server, which increases data traffic in network. Thus, client based deduplication typically communicate with a server, and Figure 12 displays how client based deduplication works with the help of a server. The client splits a file into chunks ($c1$ and $c2$) and compute indexes ($h(c1)$, $h(c2)$). Then, the client sends all the indexes of the file to server which then returns indexes ($h(c2)$) of unique chunks that have not been saved previously. The client then can send only unique chunks ($c2$) in this manner.

LBFS [54] improves space savings by adding a communication protocol that sends indexes to a server before sending a real data chunk. However, it introduces latency to run the protocol. Overall, a client based deduplication system has difficulties with limited capacity of clients to perform an expensive deduplication process.

2.9 End-to-end Redundancy Elimination

End-to-end RE like WAN optimizers [9] [10] [65] removes redundant network traffic at two end points (e.g. branch to headquarter and data center to data center). Figure 13 illustrates how end-to-end RE works. End-to-end RE like WAN optimizer is located just before an ingress router (sending side), and just after an egress router (receiving side). Suppose clients send the same files ($f1$ and $f2$) to a server. When a unique file $f1$ is transferred, the file is split into chunks (here $c1$ and $c2$) and corresponding indexes ($h(c1)$ and $h(c2)$) are saved into the cache; subsequently, chunks and indexes are saved onto disk that is now shown here. The file is compressed and delivered to server side where chunks

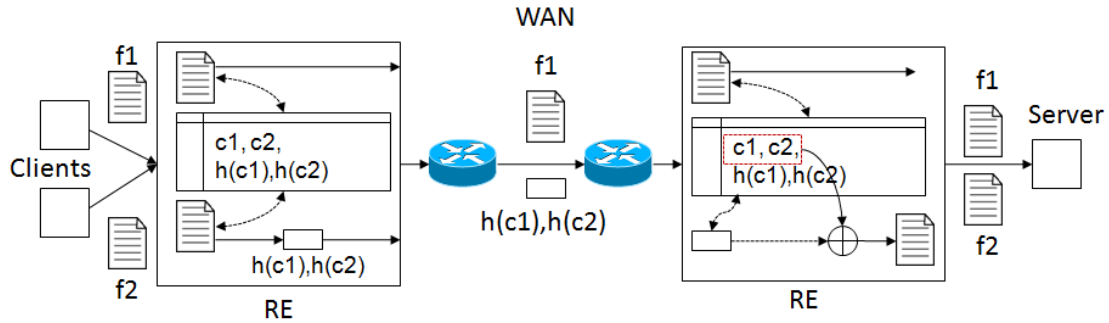


Figure 13: End-to-end redundancy elimination: $c1$ and $c2$ are chunks. $h(c1)$ and $h(c2)$ are hash keys (indexes) of chunks.

and indexes of the received file are saved into the cache.

Now, when another client sends the same file ($f2$), chunks of $f2$ are split and indexes of the chunks are compared with previously saved indexes. The file $f2$ is found to be duplicate because same indexes $h(c1)$ and $h(c2)$ are found in cache. Thus, the contents of the file are replaced (or encoded) by small sized indexes $h(c1)$ and $h(c2)$, which reduces packet size. When the encoded packet arrives at server side, a file $f2$ is reassembled with chunks $c1$ and $c2$ based on indexes in the packet. The reassembled file is directed to a destined server.

2.10 Network-wide Redundancy Elimination

Network-wide RE [4] [5] [71] eliminates repeating network traffic across network elements such as routers and switches. Network-wide RE computes indexes [64] for the incoming packet payload, and eliminates redundant packets by comparing indexes with the packets saved previously. Redundant payload is encoded by small sized shims and decoded before exiting networks. However, this approach suffers from high processing

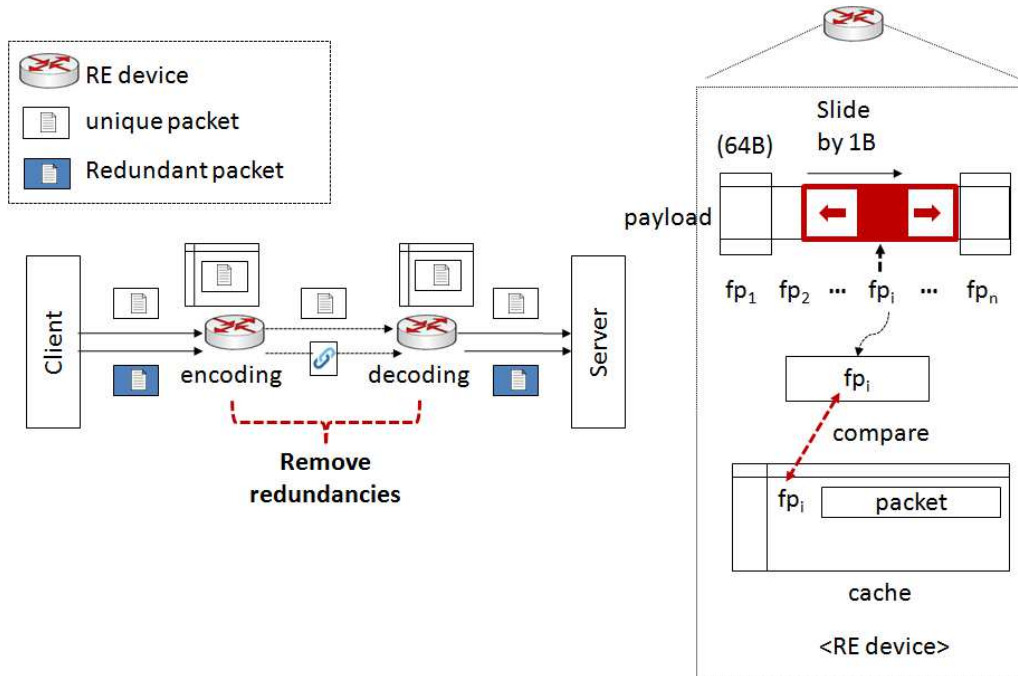


Figure 14: Network-wide redundancy elimination

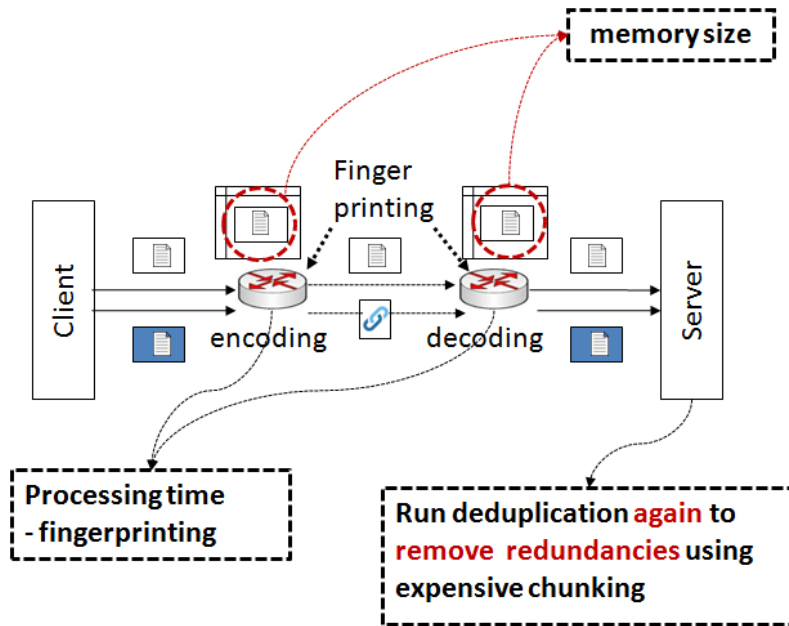
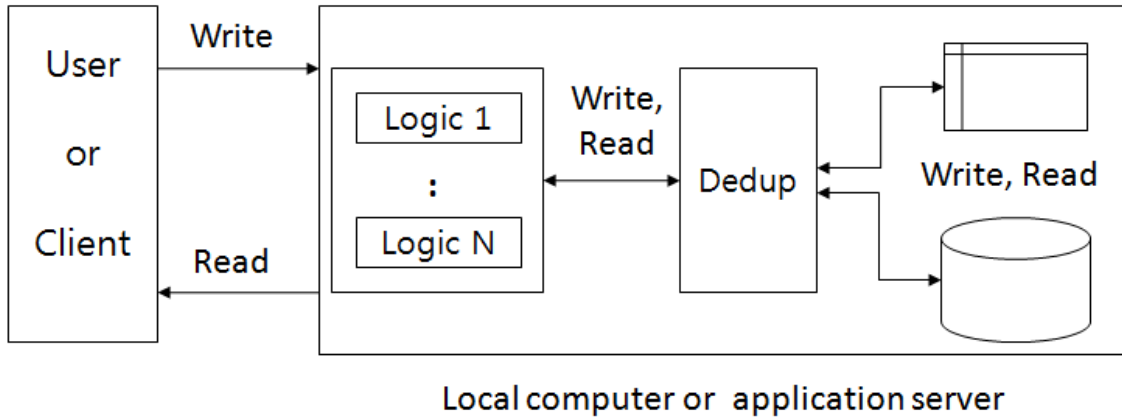


Figure 15: Network-wide redundancy elimination: issue

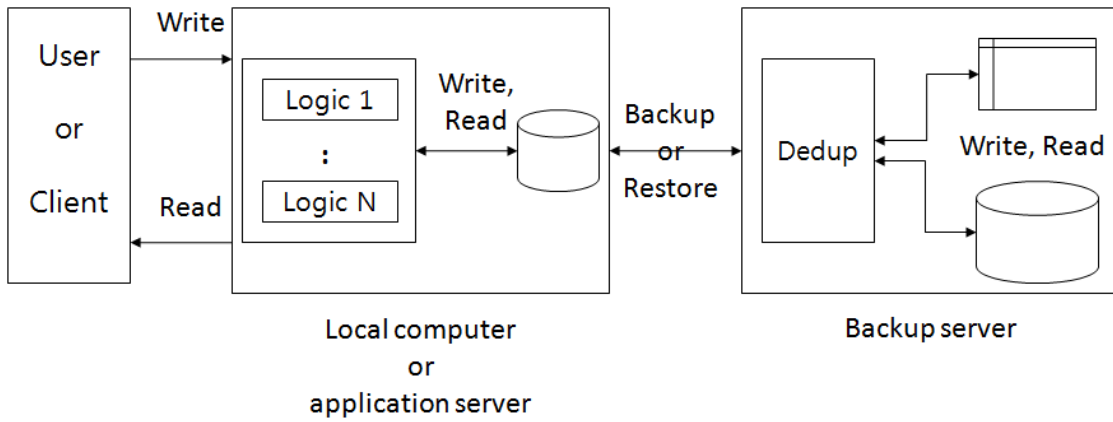
time due to sliding fingerprinting at routers and high memory overhead to save packets and indexes.

The goal of network-wide RE is to remove redundancies of packet payloads and the granularity is byte strings in a payload. Figure 14 displays how network-wide RE works. In network-wide RE, there are special routers (or switches), called RE devices. When an RE device receives a packet, it slides a small sized window on the payload, and computes fingerprints of all windows. Then, some fingerprints are compared with fingerprints in local cache. If they are same, pointed byte regions are expanded to the left and to the right while comparing with a packet in local cache. Expanded byte region is replaced by a small sized shim header with a fingerprint and byte offsets. These processes are encoding. The encoded payload is reconstructed by an RE device on a path, called decoding. Decoded packets are delivered to a server.

As we see here, network-work RE saves bandwidth in links between an encoder and a decoder. However, as shown in Figure 15, sliding fingerprinting requires excessive processing time, and packets which are saved in cache increase memory requirements. More importantly, redundancies removed in network are restored in a decoder before reaching the server. Thus, the server should run deduplication again to remove redundancies using expensive chunking. That is, there are redundant de-duplication operations in network as well as in server. We address this issue by developing SoftDance in chapter 6.



(a) Inline deduplication for primary workloads



(b) Inline deduplication for secondary workloads

Figure 16: Inline deduplication

2.11 Inline deduplication

Inline deduplication is a deduplication that removes redundancies before data are stored onto disk. Inline deduplication can be applied for primary workloads like email, user directories, databases, and secondary workloads like archives and backups. Figure 16 elaborates how inline deduplication works for primary workloads (latency sensitive) as well as secondary workloads (throughput sensitive). For primary workloads, as shown in Figure 16(a), deduplication runs on a direct write and read path. When a user or client writes data, deduplication intercepts the data and checks for redundancies. Only unique data and indexes are saved into storage along with cache. Applications using primary workloads are highly latency sensitive; thus, deduplication typically uses in-memory cache to reduce disk I/O requests. Figure 16(b) shows how deduplication works for secondary workloads. In these workloads, deduplication runs when data are archived or backed up in backup server. Backup server does not maintain additional storage.

Inline deduplication has been proposed to remove redundancies for primary workload [20] [72] and secondary workload [11] [44] [58] [78] without incurring extra space overhead and more disk bandwidth. However, this approach requires latency overhead in a write path. iDedup [72] exploits temporal locality and spatial locality to maintain fast processing time in a write path. Content address storage (CAS) systems [23] [63] run inline deduplication because blocks are addressed by their fingerprints. A few file systems [8] [70] use inline deduplication for primary storage.

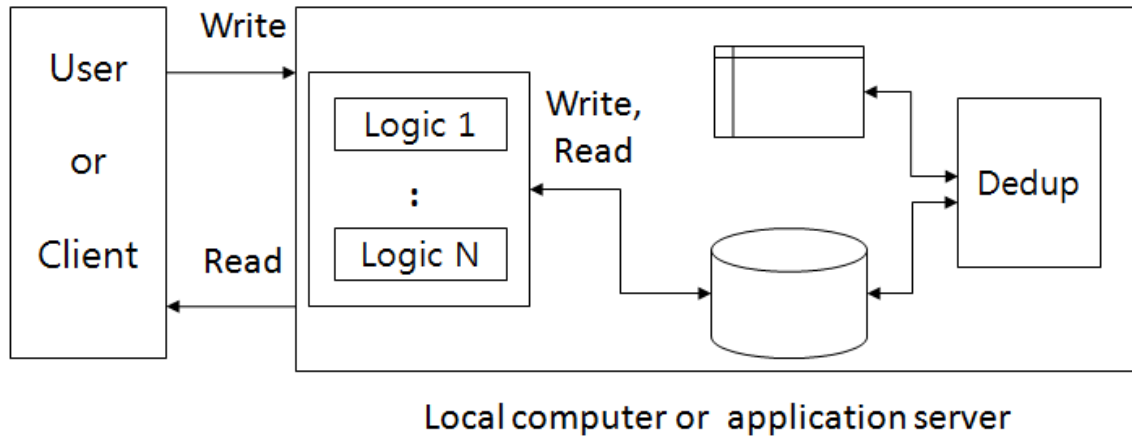


Figure 17: Offline deduplication

2.12 Offline deduplication

Offline deduplication [2] [21] [33] runs deduplication after data are stored on disk; thus, it does not involve latency overhead in a write path but requires extra storage space. As shown in Figure 17, data are saved into storage without deduplication. Offline deduplication runs out of a critical write and read path using already saved data, which does not hurt latency to write and read data. However, offline deduplication has several drawbacks: 1) extra disk space is needed to hold data temporarily before deduplication, 2) deduplication runs on system idle time, so deduplication can be very delayed if the system is running almost all the time, and 3) data on disk are loaded to memory for deduplication, so disk bandwidth is unnecessarily consumed.

CHAPTER 3

EXISTING DEDUPLICATION APPROACHES

In this chapter, we elaborate existing and representative deduplication approaches based on deduplication classification.

3.1 File-level Deduplication

File-level deduplication is used for Microsoft Exchange 2003 and 2007 based on Single Instance Store (SIS) [7]. SIS stores file contents into ‘SIS Common Store’. In SIS, a user file is managed by an SIS link that is a reference to the file called ‘Common Store File’. Whenever SIS detects duplicate files, SIS links are created automatically and file contents are saved into the common store. SIS consists of a file system filter library that implements links and a user level service detecting duplicate files (which are replaced by links). SIS can find duplicate files but cannot find large redundancies within similar files. We addressed this issue by developing the ‘Hybrid Email Deduplication System’ [40].

File-level deduplication is used for popular cloud storage systems such as Just-Cloud [39] and Mozy [52] to reduce latency in a client. Cloud storage system client applications run file-level deduplication that computes an index (hash key) of each file and checks if the index exists in a server. If the server has the index, client does not send the duplicate file. Running the file-level deduplication in the client before sending data to server allows cloud storage systems to consume less storage space and bandwidth. A

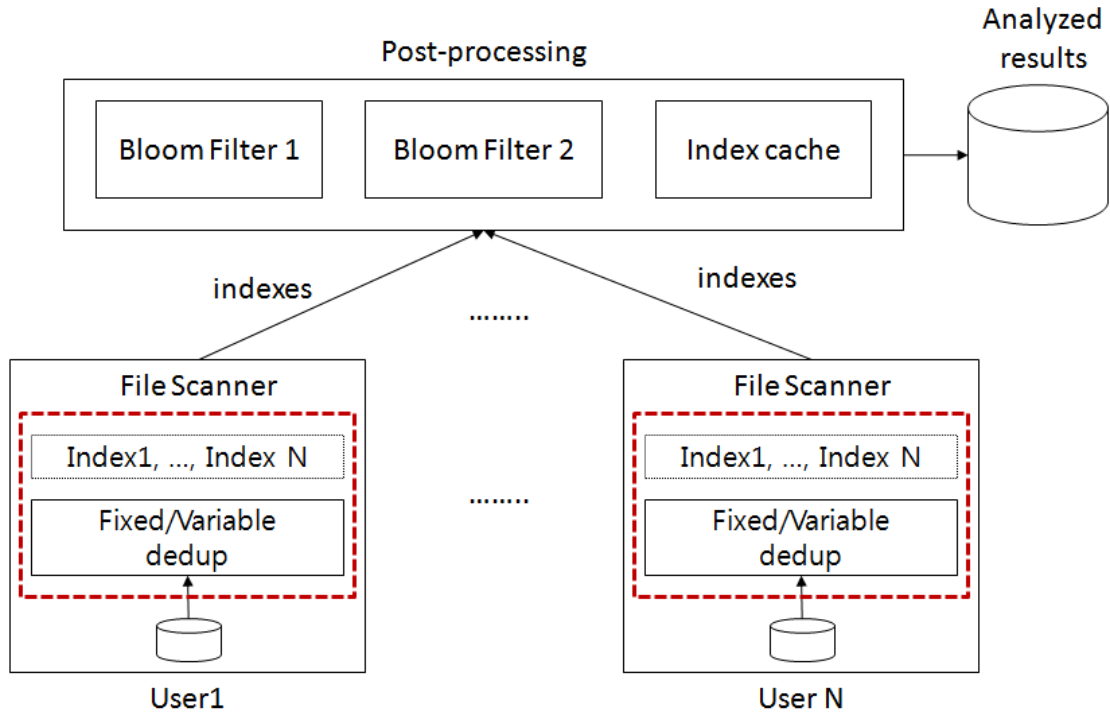


Figure 18: A study of practical deduplication: evaluation setup

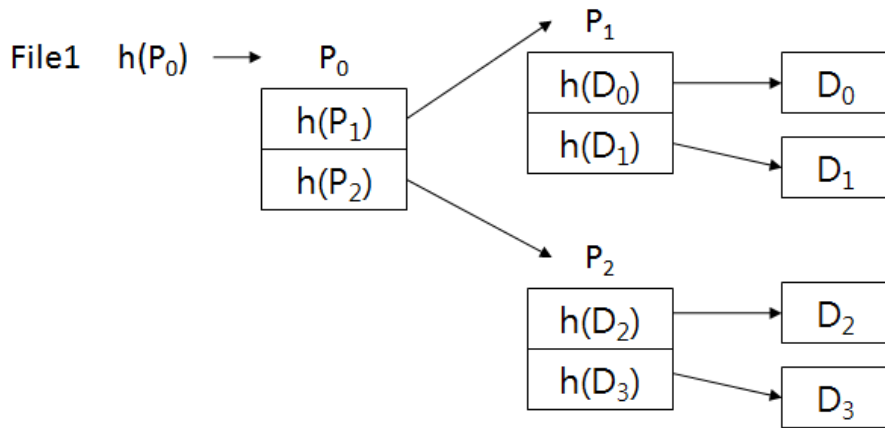
study [32] measured performance of several cloud storage systems including Mozy.

A study [46] evaluates the tradeoff in space savings between file-level deduplication and block-based (fixed-size and variable-size) deduplications, claiming that file-level deduplication provides simpler complexity and reduces more file-fragmentation than block-based deduplications. The study collected file system contents from almost 1000 desktop computers in a corporation, and measured file redundancies and space savings. Authors show file-level deduplication achieves 87% of space savings compared to block-based deduplications. Figure 18 shows the evaluation setup of the study. ‘File system scanner’ computes indexes of blocks or chunks by running fixed-size and variable-size

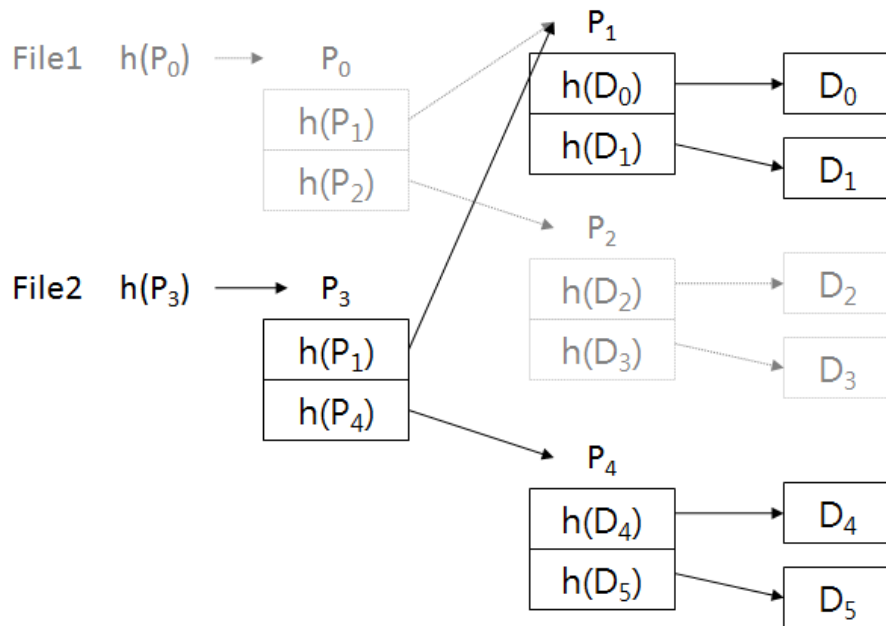
block deduplication with the minimum and maximum chunk sizes 4 KB and 128 KB respectively. The expected chunk size ranges from 8 KB to 64 KB. The computed indexes are collected by post-processing module that checks redundancies of indexes using two bloom filters. The size of the bloom filter is 2 GB. Analyzed results are saved into a database. The computed total size of files is 40 TB and the number of files is 200 million files. The file duplicates are found in post-processing by identifying files where all chunks matched. This study also mentions semantic knowledge of file structures will be useful to reduce redundancies with less overhead, and our approach ‘Structure aware file and email deduplication (SAFE)’ exploits the semantic information of file structures as shown in chapter 5.

3.2 Fixed-size Block Deduplication

Venti [63] is a fixed-size block deduplication and uses a write-once policy, preventing data from being inconsistent or malicious data loss. The main idea is that a file is divided into several blocks, and that the index (hash key) of each block is created by SHA1 hash function. If the index of the block is same as an index previously saved, the block is not saved. The index is arranged into a hash tree for reconstructing a file which contains the block. To improve the performance, Venti uses three techniques called caching, striping, and write buffering. Block as well as index are cached. Venti shows the possibility of using hash to differentiate each block in a file. Most of de-duplication applications which have been published split a file into several blocks (or chunks) and save each block based on the index (hash key) of each block.



(a) Tree structure of an original file (File1). File1 consists of four data blocks including $D_0, D_1, D_2,$ and D_3 .



(b) Tree structure of a similar file (File2). File2 consists of four data blocks including $D_0, D_1, D_4,$ and D_5 .

Figure 19: Venti tree structure of data blocks [63]

Figure 19 shows how files are saved into tree structure of Venti. A data block is pointed by an index (hash key) of the block, and the indexes are packed into a pointer block with pointers. As shown in Figure 19(a), Venti creates a hash key of a pointer block P_0 that is a root pointer block of *file1*. Venti creates new pointer blocks P_1 and P_2 that subsequently point to D_0 , D_1 , D_2 , and D_3 . Thus, data blocks of file1 are retrieved following on the tree structure of pointer blocks starting from P_0 . Figure 19(b) demonstrates how the tree structure is changed when a similar file (*file2*) is saved. Suppose *file2* has two identical data blocks (D_0 and D_1) as file1, but two unique data blocks (D_4 and D_5). Venti does not change pointer blocks but instead creates new pointer blocks (P_3 and P_4) for *file2*. *File2* can be retrieved using pointer blocks P_3 , P_1 , and P_4 .

Dropbox [14] uses fixed-size block deduplication with a 4 MB fixed block as its granularity. A study [13] discovers internal mechanisms of Dropbox by measuring and analyzing packet traces between clients and Dropbox servers. Dropbox is accessed by Web UI (<http://www.dropbox.com>) or dropbox client. We leverage SAFE into a Dropbox client to deduplicate structured files in a client side. Dropbox consists of two type of servers; one is a control server and the other is a storage server. Control servers hold meta data of files such as hash value of individual blocks and mapping between a file and its blocks. Storage servers contain unique blocks in Amazon S3 [3]. Dropbox client synchronizes its own data and indexes with Dropbox servers.

Figure 20 shows how Dropbox works. Circles with numbers are the order in which a file is saved. *File-A* is a file and *Blk-X* is a block which is separated from a file. $h(\textit{Blk-X})$ means hash value of a block. Thick $h(\textit{Blk-X})$ and $\textit{Blk-X}$ are considered as

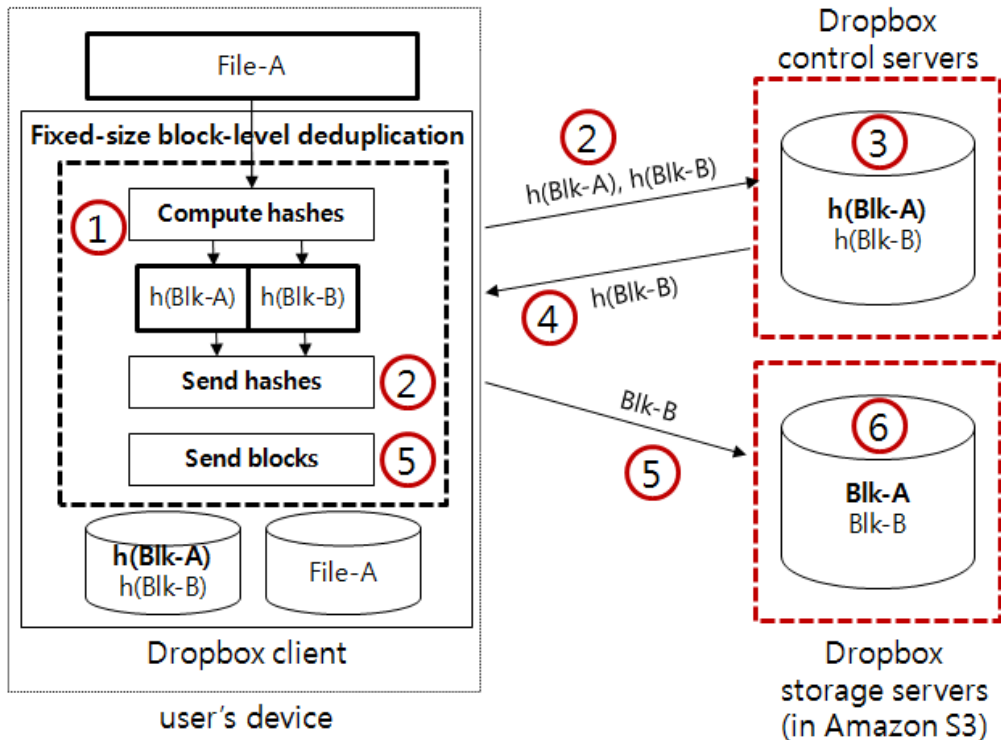


Figure 20: Dropbox internal mechanism

hash values and blocks which already existed before a file is saved. A user's device is a mobile phone, tablet, laptop, or desktop. Dropbox follows the next steps to save a file.

(1) As soon as a user saves *File-A* into a shared folder in a Dropbox client, the fixed-size block deduplication of Dropbox splits the file into blocks based on 4 MB granularity, and computes hashes of the objects. If a file is larger than 4 MB, a file is the same as an object, and an hash value of the file is computed. Dropbox uses SHA256 [57] to compute a hash value. (2-4) Dropbox client sends computed all hash values of a file to a control server that returns only unique hash values after checking previously saved hash values. In this example, hash of *Blk-B* is returned to a client because the hash of *Blk-A* is found to

be duplicate. (5-6) The Dropbox client sends to the storage server the blocks of returned indexes. Ultimately, storage servers have unique blocks across all Dropbox clients. Note that storage saving occurs in a server (thanks to not saving *Blk-A* again), and the incurred network load is reduced thanks to sending *Blk-B* only.

3.3 Variable-size Block & Server-based Deduplication

Variable-size block deduplication involves expensive chunking and indexing for finding large redundancies, requiring an efficient in-memory cache and on-disk layout in high capacity servers. DDFS [78] exploits three techniques to relieve disk bottleneck, reducing processing time. The ‘Summary vector’ that is a compact in-memory data structure is used to find new data. ‘Stream-informed segment layout’, on-disk layout, is used to improve spatial locality for both data and indexes. The idea of Stream-informed segment layout is that a segment tends to reappear in the similar sequences with other segments. This spatial locality is called ‘Segment duplicate locality’. ‘Locality preserved caching’ uses segment duplicate locality to acquire a high hit ratio in the memory cache. The study removes 99% of the disk accesses and achieves 100 MB/sec and 210 MB/sec for single-stream throughput and multi-stream throughput respectively.

Sparse indexing [44] uses sampling and a sparse index to reduce the number of indexes, decreasing RAM requirements. The study chooses small portions of chunks in the byte stream as sample, and avoids full chunk indexes unlike DDFS. This approach employs *chunk locality*, tendency for chunks in backup data streams to reoccur together.

Figure 21 shows the deduplication process of Sparse indexing. In Sparse indexing, segment is the unit of storage and retrieval, and a sequence of chunks. A byte stream is split to chunks by Chunker using variable-size chunking, and a sequence of chunks becomes a segment by Segmenter. Two segments are similar if they share a number of chunks. Champion chooser chooses sampled segments, called champion, from sparse index (in-memory index). Deduplicator compares chunks in incoming segments with chunks in champions (selected segments). Unique segments are saved to sparse index for future comparison, and new chunks are saved into Container store.

3.4 Hybrid Deduplication

Hybrid approaches have been proposed by adaptively using variable-size block-level deduplication and file-level deduplication either based on fixed policy or dynamically changed file information [40] [50]. Min et al. [50] employs context-aware chunking where they use a file-level deduplication for multimedia content, compressed files, or encrypted content and uses variable-size block-level deduplication for text files. Our approach, Hybrid Email Deduplication System (HEDS) [40] first separates the message body and individual attachments, and performs a variable-size block-level deduplication if the object size exceeds a predefined threshold. Otherwise, a file-level deduplication is used.

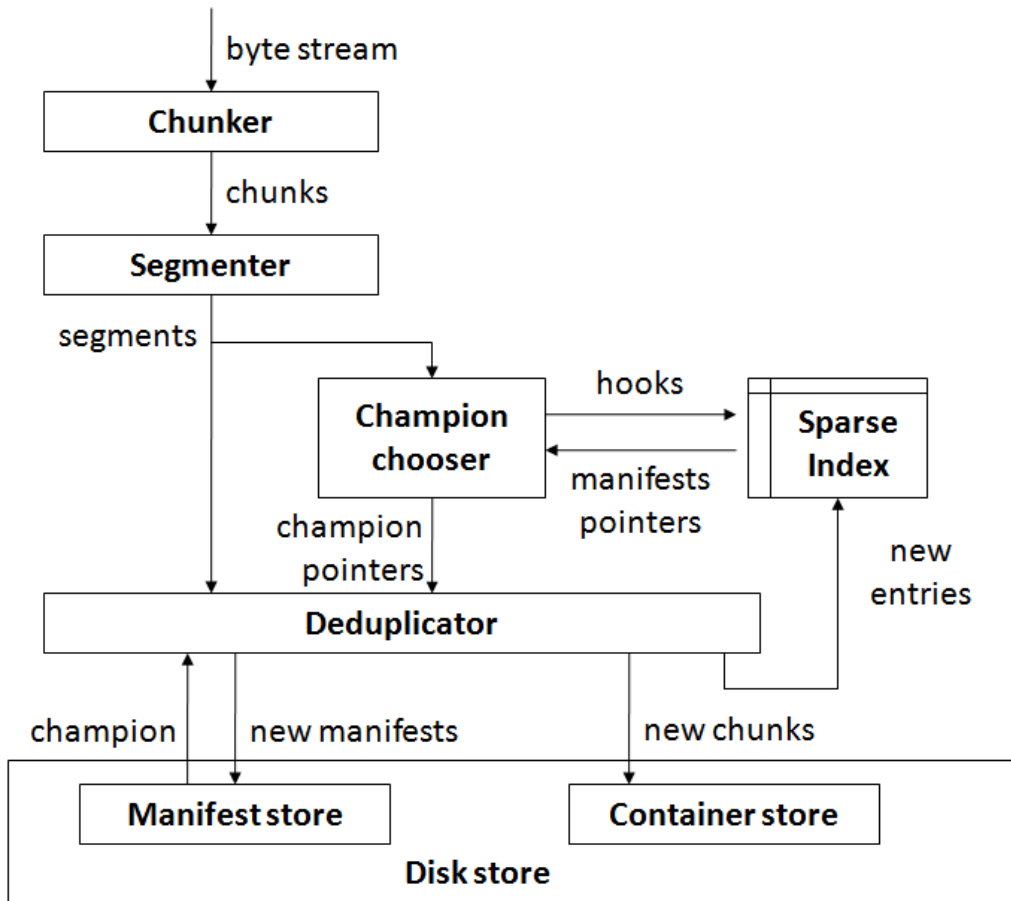


Figure 21: Sparse indexing: deduplication process [44]

3.5 Object-level Deduplication

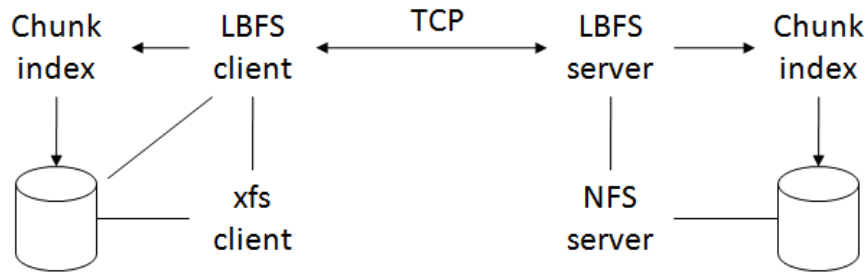
Fixed-size block deduplication and variable-size block deduplication can be used for all types of files because they rely on the physical byte-string format of a file. However, for specific file formats, they may be inefficient due to expensive chunking. Thus, object-level deduplication that splits a file based on the semantic (or logical) format of a file has been proposed. A few *structure-aware data deduplication* techniques [41] [43] [45] [77]

have been proposed to simplify the chunking mechanism by using objects. Our approach, SAFE [41] splits structure files including compressed files, document files (docx, pptx, and pdf), and emails based on files' structured formats. ADMAD [45] separates a file into variable-size semantic segments, called meaningful chunks (MCs), based on the metadata of each file. Although the idea of ADMAD to decompose a file into objects according to the object structure is similar to the SAFE, ADMAD is limited to a specific file format. For example, ADMAD does not deal with document file types such as docx, pptx, and pdf. In addition, ADMAD does not handle an email with multiple attachments. [43] and [77] show similar concepts where they deduplicate structured objects.

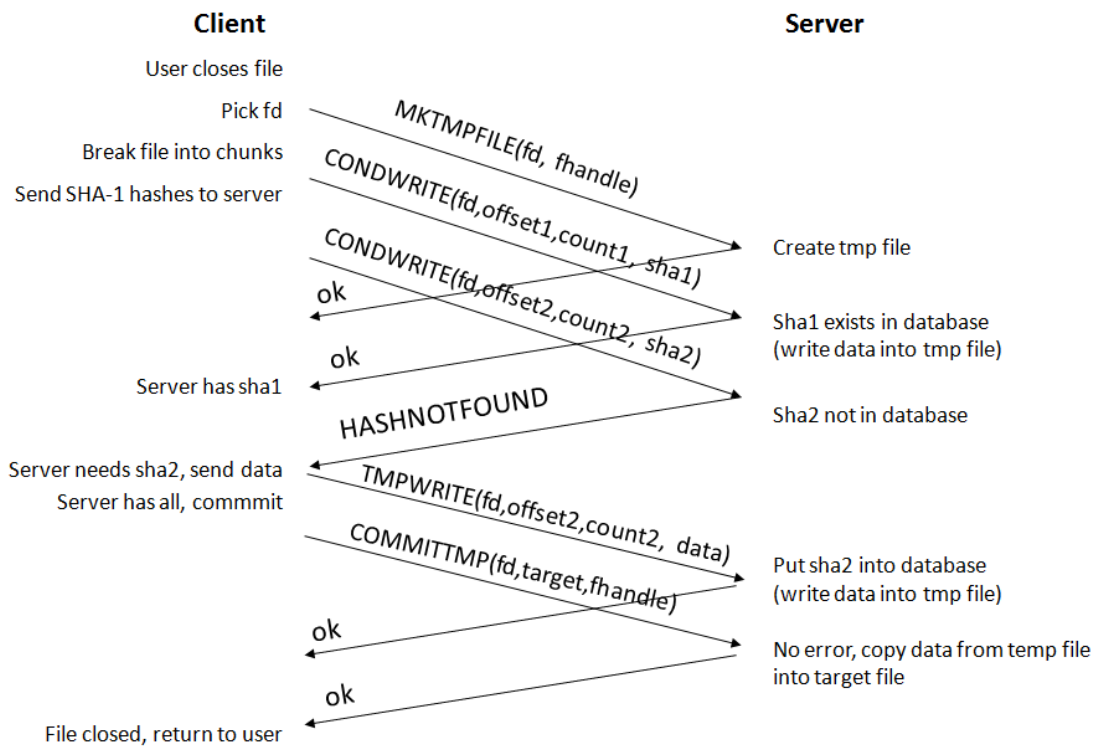
3.6 Client-based Deduplication & End-to-end Redundancy Elimination

Low bandwidth file system (LBFS) [54] reduces latency and network bandwidth through collaboration of the client and server. That is, LBFS avoids sending data over network when the same data can already be found in the server's file system or the client's cache. In order to reduce bandwidth requirement, LBFS exploits cross-file similarity. As shown in Figure 22(a), LBFS consists of LBFS client and server, and both sides maintain chunk indexes in chunk database.

Figure 22(b) shows how LBFS works when a file is written to a server from a client. When a user closes file, a client chooses a file descriptor and calls MKTMPFILE RPC; subsequently, a server creates a temporary file. A client splits a file into chunks (chunk1 and chunk2) and compute hash keys of chunks, and calls CONDWRITE RPCs with hash keys. Suppose the server has sha1 (hash key for chunk1), but does not have



(a) LBFS implementation



(b) LBFS: write a file

Figure 22: Low bandwidth file system (LBFS) [54]

sha2 (hash key for chunk2). Server returns HASHNOTFOUND for sha2 request; that is, server does not have chunk2. The client sends only chunk2 to server, and server create a file with chunk1 (previously saved chunk) and chunk2 (chunk received by TMPWRITE RPC). LBFS can be considered as a client-based deduplication because the client split the file into chunks and saves the indexes. Also, LBFS can be considered as end-to-end redundancy elimination because client and server holds same chunks and indexes, only unique chunks are transferred through the network, and both sides (client and server) maintain chunks for unique and redundant files.

3.7 Network-wide Redundancy Elimination

A study [4] proposes network-wide deployment of redundancy elimination technology. Authors assume that routers have the ability to detect and encode redundant content from network packets on the fly by comparing packet contents that were stored in a cache previously. In this approach, unique packets and corresponding fingerprints of bytes in packet payload are saved into a packet store and fingerprint store. When a packet comes to a router, a small sized window slides on the payload in a packet, and fingerprints are computed for all windows. Among all fingerprints, representative fingerprints are selected randomly. If the same fingerprints are found in the cache, the matched region from pointed byte regions on a payload are expanded both to the left and to the right while comparing the two packets (incoming packet and packet in cache). The expanded region is replaced by a small sized shim header.

Figure 23 illustrates how many redundant packets are removed. Figure 23(a)

shows traditional shortest path routing where 18 packets are transferred from a sender to two destinations D_1 and D_2 . Using redundancy elimination on the routers, packet P_1 on each link is removed as shown in Figure 23(b), which is a 33% reduction of total packets. This study proposes redundancy-aware routes based on redundancy profile (that explains how oftent content is replicated across different destinations) for intra domain routing and inter domain routing. Figure 23(c) supports the idea that redundancies are further reduced using redundancy-aware routing, which amount to a 44% reduction of total packets.

3.8 Inline & Offline Deduplication

Inline deduplication [16] [11] runs deduplication before data are saved onto disk storage. iDedup [72] has been proposed as inline deduplication for primary workload. iDedup exploits spatial locality and temporal locality to gain performance (running time). For spatial locality, iDedup performs selective deduplication and mitigates the extra seek time for sequentially read files. For this purpose, iDedup examines blocks at write time, and only deduplicate full sequences of file blocks if and only if the sequence of blocks are 1) sequential in the file and 2) have duplicates that are sequential on disk. For temporal locality, iDedup maintains dedup-metadata as a *Least Recent Used (LRU)* cache by which iDedup avoids dedup-metadata IOs.

ChunkStash [11] is a flash-assisted inline deduplication system where chunk metadata (with chunk index as key, and with chunk location and length as value) are saved into flash memory rather than disk. Considering that flash memory is 50 times faster than disk,

ChunkStash reduces the penalty of index lookup misses in RAM, which increases inline deduplication throughput. ChunkStash also uses in-memory hash tables using the variant of cuckoo hashing [61], and compact key signatures rather than full keys are stored in the hash table, which reduces RAM size.

HYDRAsTOR [16] is a grid of storage nodes. It works based on a distributed hash table (DHT) to save blocks into distributed storages, inline de-duplication based on immutable and content-addressed and variable-sized blocks, data resilience by erasure coding, load balancing, preservation of locality of data streams by pre-fetching. HYDRAsTOR achieves scalability (by DHT), efficient utilization (by deduplication), fault-tolerance (by data resiliency), and system performance (by load balancing, locality, and prefetching).

CHAPTER 4

HEDS: HYBRID EMAIL DEDUPLICATION SYSTEM

In this chapter, we show a server-side deduplication component, HEDS (Hybrid Email Deduplication System) for the proposed deduplication framework. HEDS removes redundancies by trading-off of file-level and block deduplication for email systems while achieving good storage space savings and low processing overhead.

4.1 Large Redundancies in Emails

Email is a prominent method of communication today, and the volume of emails is greatly increasing and requires huge storage space on email servers. Email servers have large amount of redundancies. For example, an email with multiple recipients is copied into multiple mailboxes, and email threads (where emails on the same topic are repeatedly sent and received with same or similar attachments) increase redundant attachments. The redundancies in the emails are further increased as they are copied over multiple storages for reliability or performance.

The volume of email data can be reduced by properly removing the redundancies. Fixed-size and variable-size block deduplication can be used to find redundant contents in emails. However, fixed-size block deduplication cannot find redundancies between similar emails whose beginning contents are changed due to the offset shifting problem.

Variable-size block deduplication efficiently finds redundancies of similar emails but increases processing time overhead due to expensive chunking by sliding a window. File-level deduplication can be used to find redundancies of duplicate emails with multiple recipients quickly, but cannot find redundancies inside emails and attachments, resulting in low space savings.

There have been few studies to remove redundancies found in email. Single Instance Store (SIS) [7] uses file-level deduplication where an email is the unit of compared data. In this approach, only unique email is saved, and redundant emails are linked by pointers, which increases storage space savings by not saving the same emails. However, SIS does not exploit redundancies within email messages and attachments.

Considering the overhead as well as performance of a deduplication, we developed Hybrid Email Deduplication System, HEDS, that trades-off of file-level and variable-size block deduplication in terms of space savings and index overhead. HEDS separates attachments from an email, runs file-level deduplication for the message body and separated attachments, and adaptively runs variable-size block deduplication only if data size is over a predetermined threshold. The reason for this threshold is that small sized message body and attachments are generally unique, and that using block deduplication for small data does not give any performance benefits considering the processing overhead. Evaluation of results using real email datasets show that HEDS achieves a good deduplication ratio while keeping the CPU and memory overhead manageable. Therefore, it sheds light on in-line deduplication for email servers. This chapter is organized as follows. We describe the design and implementation of HEDS in Section 4.2. The evaluation results are shown

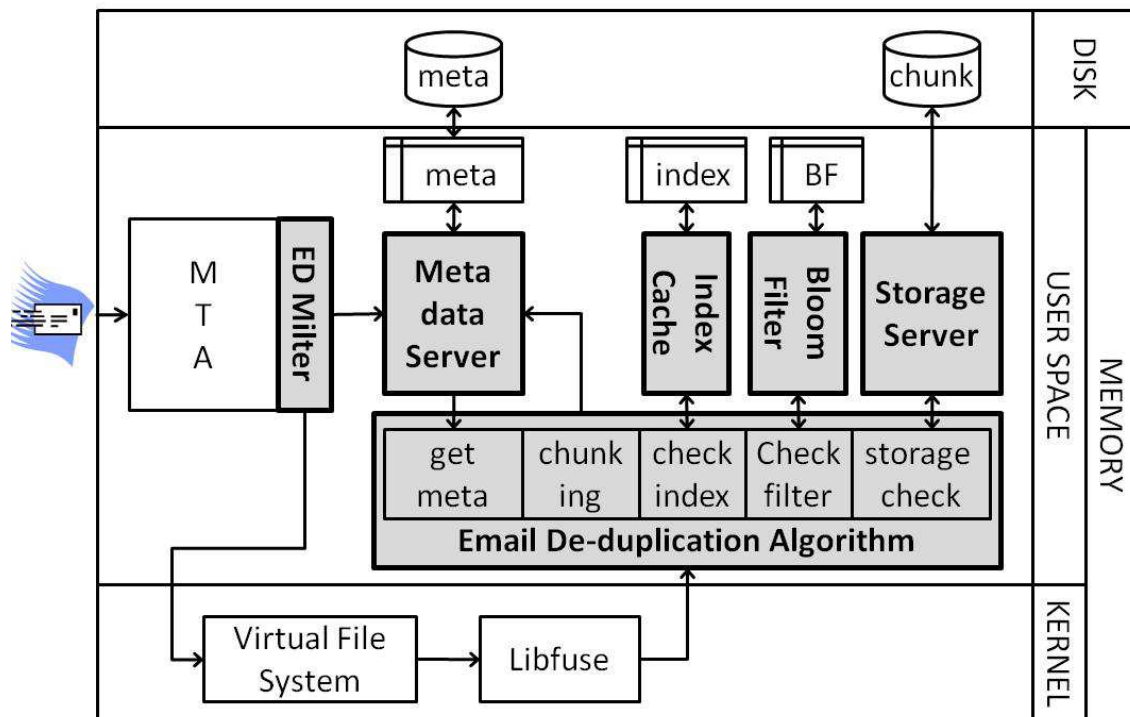


Figure 24: Proposed hybrid email deduplication system (HEDS)

in Section 4.9. We conclude this chapter in Section 4.13.

4.2 Hybrid System Design

To explain the architecture of HEDS, we begin by presenting an overview of HEDS, and then elaborate each module in HEDS. HEDS is a server-based deduplication that consists of six modules including EDMilter, meta data server, chunk index cache, bloom filter, storage server, and email de-duplication algorithm (EDA). When an email comes in the Mail Transfer Agent of a receiving Sendmail server, EDMilter intercepts the email and divides the email into meta data and content. Content consists of message body

and attachments. EDMilter forwards the content to the Email De-duplication Algorithm (EDA) through a virtual file system, and delivers the meta data to the meta data server. The meta data server holds meta data such as email id, recipients, and sent date. EDA deduplicates content that is intercepted by `Libfuse` through a virtual file system. EDA plays a key role in the deduplication and communicates with all other modules. We implemented EDA with filesystem on userspace, FUSE [28]. Chunk index cache and bloom filter speed up processing time by reducing the number of disk accesses. We explain each module hereafter.

4.3 EDMilter

As shown in Figure 25, we have developed the Email Deduplication Milter, called the EDMilter, based on the Milter [49] API. Milter is an email filter that intercepts emails coming into the sendmail server. When a sendmail server receives an email, the Milter Library accepts an email from the Mail Transfer Agent (MTA) and passes the email into the EDMilter with a callback. The EDMilter extracts needed metadata from the SMTP header such as a mail id, senders, recipients, the number of recipients, and the size of email content that comprises of the body and attachments. At the same time, the EDMilter receives the content from the email and requests to save it into a directory that is a mounting point in a virtual file system. EDMilter also sends the email meta data to the meta data server through a message queue.

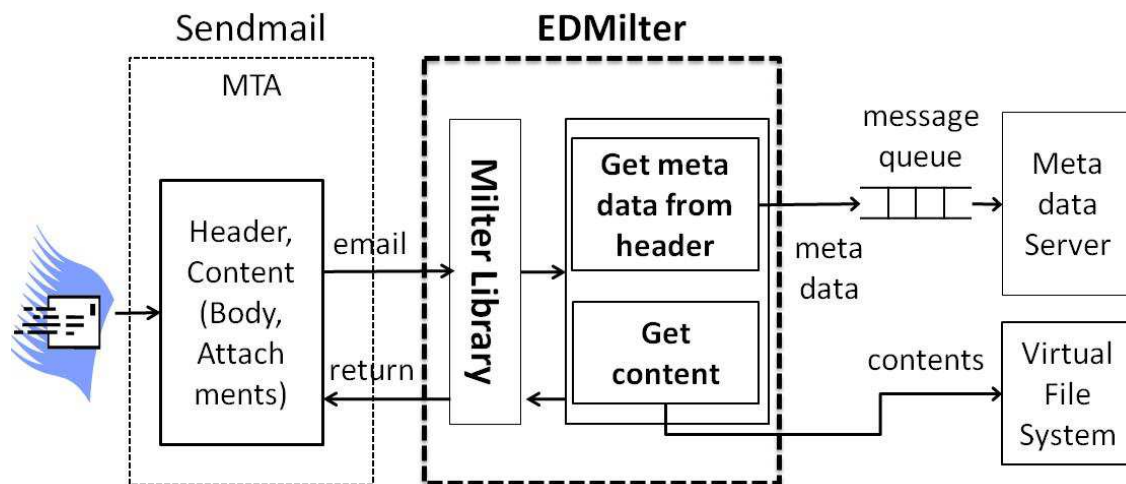


Figure 25: EDMilter

4.4 Metadata Server

The metadata server saves email meta data and chunk indexes for each email. Email meta data includes email id that is 14 byte strings, recipients, the number of recipients, and the size of email contents. The chunk indexes are received from the Email Deduplication Algorithm (EDA) when EDA splits an email content to chunks if the size of the content is over the threshold. Ultimately, the metadata server saves the meta data and chunk indexes into a meta data store. Meanwhile, to speed up reading and writing based on temporal locality, the metadata server maintains meta data and chunk indexes of the latest emails in a meta data cache. Each meta data in the cache has a time stamp to evict old meta data based on Least Recently Used (LRU) in case the cache grows over the cache size limit. The size of the metadata cache is configurable.

4.5 Bloom Filter

A Bloom filter is used to see if duplicate chunks of a current email exist in chunk storage. The Bloom filter is a bit array of m bits, that are set to 0 initially. Given a set U , each element u ($u \in U$) of the set is hashed using k hash functions h_1, \dots, h_k . Each hash function $h_i(u)$ returns an array index in the bit array, that ranges from 0 to $m - 1$. Subsequently, a bit of the index is set to 1. If the bit of the index was set to 1, it stays 1. The Bloom filter is used to check if an element was already saved into a set. When an element attempts to be added into the set, if one of the bits corresponding to the return values of hash functions h_1, \dots, h_k is 0, the element is considered as a new one in the set. If bits corresponding to return values of hash functions are all 1, the element is considered to exist in the set. However, the Bloom filter has a false positive where it says that an element exists though it does not. In HEDS, the purpose to use the Bloom filter [6] is to speed up the writing of emails or chunks by reducing the number of disk accesses. DDFS (Data Domain File System) [78] shows how big the memory size for Bloom filter should be used with a certain false positive rate; for example, to achieve a 2% false positive, the smallest size of the Bloom filter is $m = 8n$ bits ($m/n = 8$), and the number of the hash functions can be 4 ($k = 4$) where m is the size of the Bloom filter in bits, n is the key size. We use four hash functions, and the key size is 160 bits, which is the size of the SHA1 [56] hash key. Therefore, the smallest size of the bloom filter for our case is $8 * 160$ bits = 1280 bits.

4.6 Chunk Index Cache

Chunk index cache is the next level of cache after Bloom filter, and saves indexes for chunks that are saved as unique chunks. The chunk indexes are classified into three different categories based on what the source of the chunk is. The source can be an entire email, an attachment, or part of an email text. Chunk indexes of latest emails are saved into the chunk index cache, and if chunks or a chunk of a current email exists in the chunk index cache, the chunk(s) is not saved into the chunk store on a disk. Because of limited size of the chunk index cache, old indexes are evicted to chunk storage if size of cache grows over a certain threshold. Likewise, loaded indexes from the chunk store have new time stamps. The key of each entry in chunk index cache is a SHA1 hash value of a chunk.

4.7 Storage Server

The storage server checks whether or not the chunks of the current email exist in chunk storage by accessing the disk, saving non-existent chunks into the chunk storage, or reading chunks from chunk storage. We have used BerkeleyDB for chunk storage, and pairs of <chunk index, chunk> are saved into the chunk storage.

4.8 EDA (Email Deduplication Algorithm)

The Email Deduplication Algorithm (EDA) interacts with all other modules in HEDS. Algorithms 1, 2, and 3 show how EDA works on an email with or without attachments. As shown in Algorithm 1, EDA separates email contents into message body and attachments that are further divided into individual attachment. As shown in Algorithm 2,

Algorithm 1 Email Deduplication Algorithm

Input: email content

```
1: if Not Exist(attachments) then                                ▷ email without attachments
2:   messageBody ← email content
3:   HybridDedupDecision(messageBody)
4: else                                                            ▷ email with attachments
5:   separate email content into message body, attachments
6:   HybridDedupDecision(messageBody)
7:   for all each attachment ∈ attachments do
8:     HybridDedupDecision(attachment)
9:   end for
10: end if
```

Algorithm 2 HybridDedupDecision

Input: data, size_threshold

```
1: if size(data) > size_threshold then                            ▷ variable-size block deduplication
2:   chunks ← variableSizeChunking(data)
3:   for all each chunk ∈ chunks do
4:     checkIndexAndSaveData(chunk)
5:   end for
6: else                                                            ▷ file-level deduplication
7:   checkIndexAndSaveData(data)
8: end if
```

EDA checks the size of a divided attachment or message body. If size of the data is over a configurable threshold, EDA splits the data into chunks by using variable-size chunking based on Rabin fingerprint [64]. Then, as shown in Algorithm 3, EDA checks to see if each chunk or data has been already saved based on Bloom filter and then chunk index cache. In case of chunk index cache miss, EDA checks indexes at chunk store in disk. If the Bloom filter says “no” (this means chunk or data are unique), EDA saves a chunk (or data) and the corresponding index into store. The reason to check chunk index cache

Algorithm 3 Check Index and Save Data

Input: data**Output:** saved data

```
1: if ExistInBloomFilter(data) then
2:   index ← hash(data)
3:   if ExistInChunkIndexCache(index) then                                ▷ duplicate in cache
4:     return
5:     if ExistInChunkStore(index) then                                    ▷ cache miss, duplicate in storage
6:       return
7:     end if
8:   end if
9: end if                                                                    ▷ unique data
10: index ← hash(chunk)
11: saveToStore(index, chunk)
```

after Bloom filter says “yes” (that means chunk or data may be redundant) is due to false positive of Bloom filter: the chunk or data may or may not be redundant. Also, the reason for relying on size is that small sized message body or attachments tend to be unique, and using variable-size block deduplication does not give any benefits considering overhead costs.

Figure 26 shows how variable-size block deduplication works in a case where EDA adaptively runs block deduplication. As is shown in Figure 27, EDA basically runs file-level deduplication if size is under the threshold. In this case, EDA does not separate the content into chunks but considers the content as a chunk: that is, none of variable-size chunking is necessary, which reduces index and processing overhead.

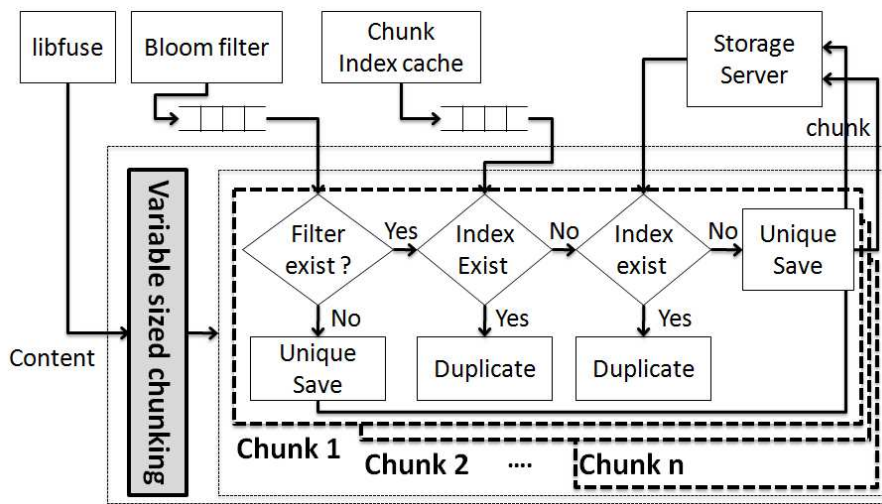


Figure 26: Block deduplication at EDA

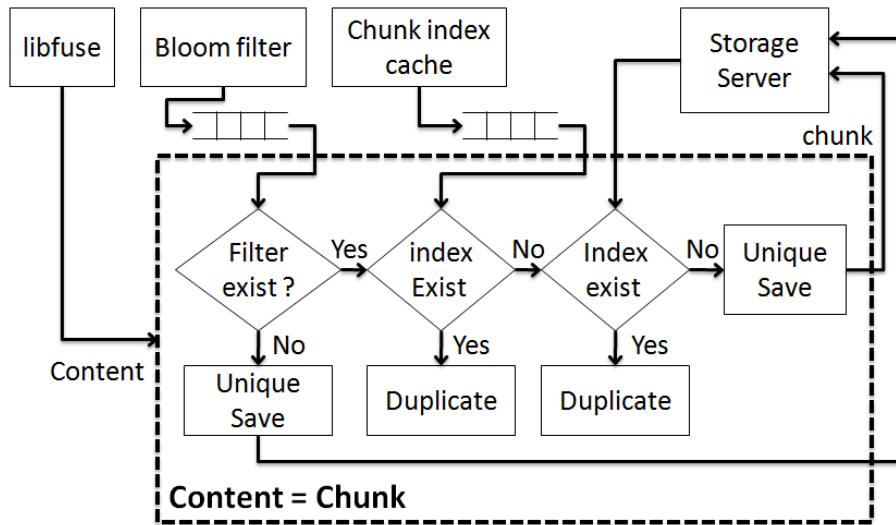


Figure 27: File-level deduplication at EDA

Table 2: Datasets

	Enron dataset	Gmail dataset
Type	Corporate emails (Enron)	Personal emails
Attachment	Removed	Retained
Number of users	150	1
Number of emails	0.5 million	0.01 million
Size of dataset	1.3 GB	1 GB
Duration	1998 - 2002	2007 - 2011

4.9 Experiment Setup

We evaluate HEDS with respect to deduplication performance, and the overhead costs of memory and CPU usage with the generated chunk indexes. We compare HEDS with file-level and variable-size block deduplications. As for the deduplication performance, we use a deduplication ratio that is computed as below.

$$\text{Dedup ratio} = 1 - (\text{deduped size} / \text{original size})$$

For the experiments with HEDS, we have set up two sendmail systems where one system sends and the other system receives emails. We also deployed an internal DNS server for the mail servers. The speed of network cards of the servers is 100Mbps. All servers have a Linux operating system whose kernel version is 2.6.35.9, and the version of mail servers is sendmail 8.14.4. We experiment with two datasets including a corporate email dataset, called the Enron dataset [42] and a single user gmail dataset. Table 2 shows the summary information of the two datasets.

For the Enron dataset experiment, we created 150 mail users for the receiving email server, according to the recipients shown in the dataset. With the gmail dataset, we created only one email user who receives all the emails, as the gmail dataset belongs to

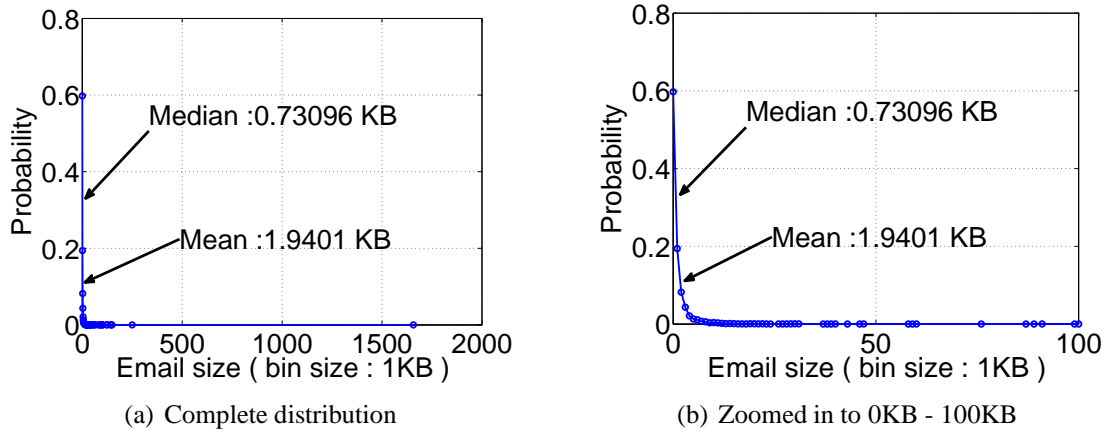


Figure 28: Distribution of the Enron corporate email sizes

one person. At a sender's email server, emails were sent one by one sequentially, in the order of emails in the datasets. For all cases, we analyze the deduplication ratios and the overhead of CPU and memory to process and store the chunk indexes.

We adjusted the threshold size, based on which HEDS performs deduplication on either an file-level or a block, adaptively. In order to gain insight on a proper threshold size, we observed the distributions of email sizes of the datasets.

Figures 28(a) and 29(a) display complete distributions, and Figures 28(b) and 29(b) are distributions with ranges zoomed in ranges for a closer look. The mean and median email sizes for the Enron dataset were about 1.9 KB and 0.7 KB, respectively. Meanwhile, the mean and median email sizes for the gmail dataset were around 28 KB and 5 KB, respectively. Note that the Enron dataset did not include attachments in the emails. Thus, we select the threshold sizes that are about or greater than the mean value of 1 KB and 2 KB for the Enron dataset. Bigger threshold sizes are chosen including 512 KB, 128 KB, 16 KB, and 4 KB for the gmail dataset. As for the expected average chunk

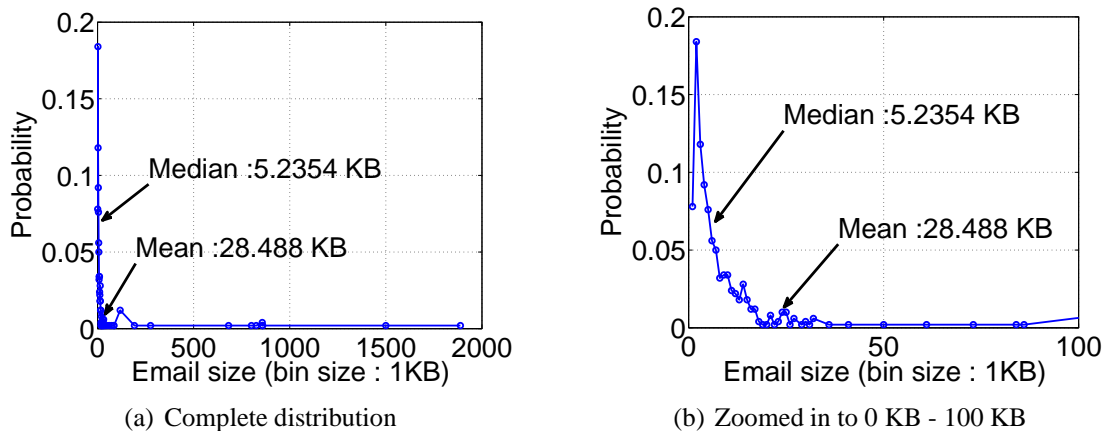


Figure 29: Distribution of the gmail personal email sizes

size, we have used 0.5 KB for the Enron dataset, and 2 KB for the gmail dataset, considering the means and medians of the datasets. We note that the previous deduplication studies [44, 46, 78] have used the expected average chunk size ranging from 4 KB to 64 KB.

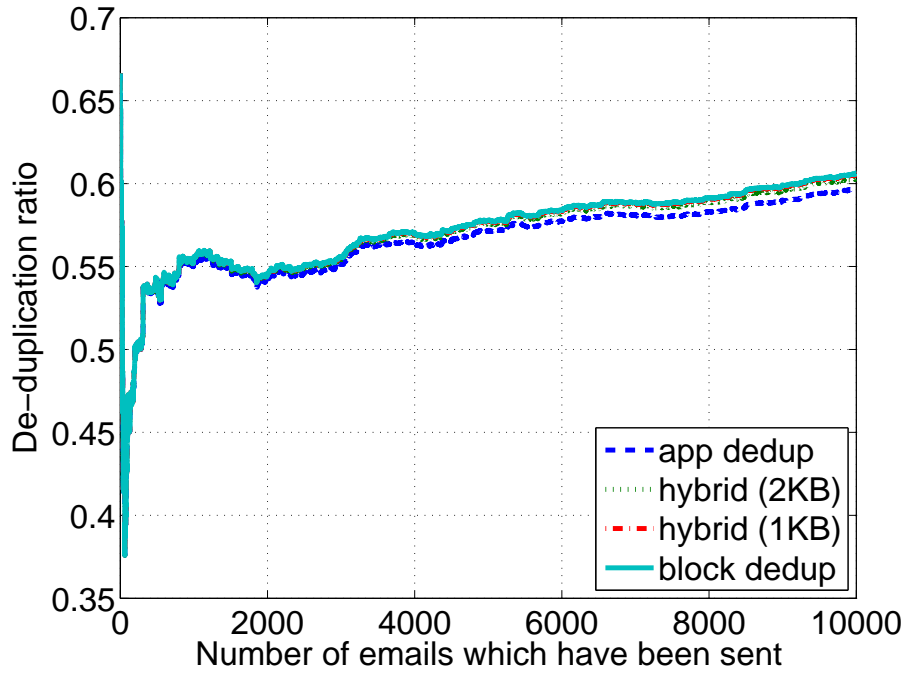
4.10 Deduplication performance

In this section, we measured a deduplication ratio that indicates how many redundancies are removed. Overall, we discovered that most of redundancies are found in attachments rather than in message bodies. Thus, for corporate (Enron) datasets without attachments, variable-size block deduplication and HEDS showed lower deduplication ratios than file-level deduplication because the low deduplication ratio cannot offset the index overhead. However, for the gmail dataset with attachments, variable-size block deduplication and HEDS has a greater deduplication ratio due to large redundancies coming from attachments. We explain in detail.

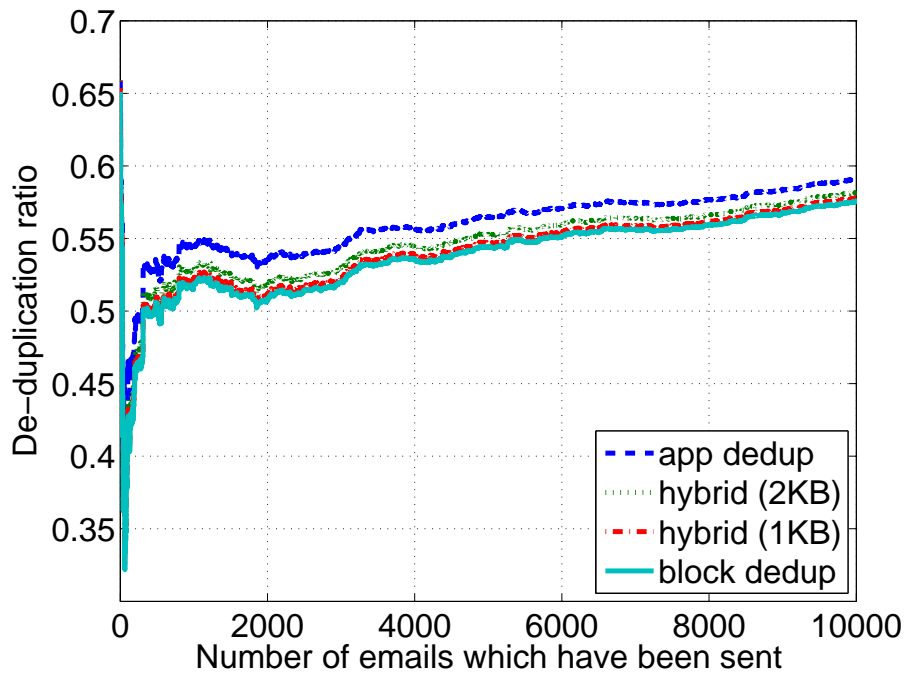
Figure 30 shows the deduplication ratio of Enron datasets. ‘App dedup’ means file-level deduplication. ‘Block dedup’ means variable-size block deduplication. ‘Hybrid’ means HEDS with variable thresholds, 1 KB and 2 KB. All deduplication showed a deduplication ratio over 55% on average. This means a sending email server sent to 2 recipients on average. Without the index as shown in Figure 30(a), block deduplication and HEDS achieved 2% to 3% more deduplication ratio than file-level deduplication. However, with the index as shown in Figure 30(b), the slight advantage of block deduplication and HEDS is overridden due to chunk index overhead.

For gmail dataset as shown in Figure 31, the deduplication ratio is different as compared to Enron datasets. Figure 31(a) shows deduplication ratios without chunk index overhead. Since the gmail dataset belongs to one person, we did not see the benefit using file-level deduplication. That is, the deduplication ratio of file-level deduplication is 0% meaning that the file-level deduplication cannot reduce any storage size. By contrast, block deduplication reduced 15% more space than file-level deduplication. HEDS with small threshold sizes like 4 KB, 16 KB has the same space savings as block deduplication, and HEDS with large threshold sizes like 512 KB reduces 10% more space savings than file-level deduplication. Figure 31(b) depicts deduplication ratios including the chunk index overhead. Even after including the overhead, the behavior of deduplication ratios shown in Figure 31(a) still remains, which indicates that the removed duplicates well over-compensate for the overhead.

We next investigated the sudden increase of deduplication ratios observed in Figure 31(a) with the gmail dataset. We found that they were caused by temporal locality of

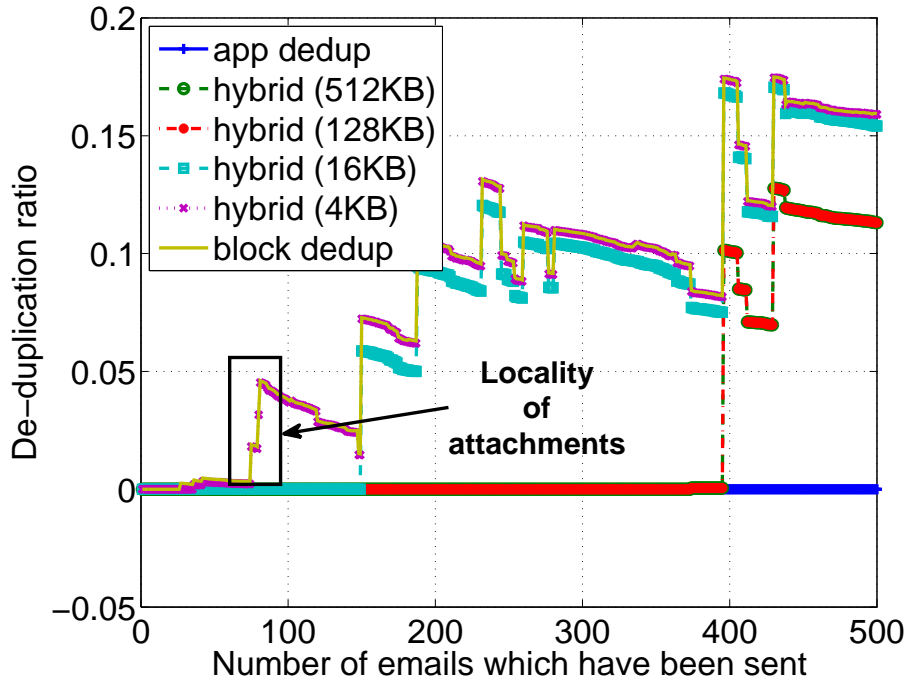


(a) Without index

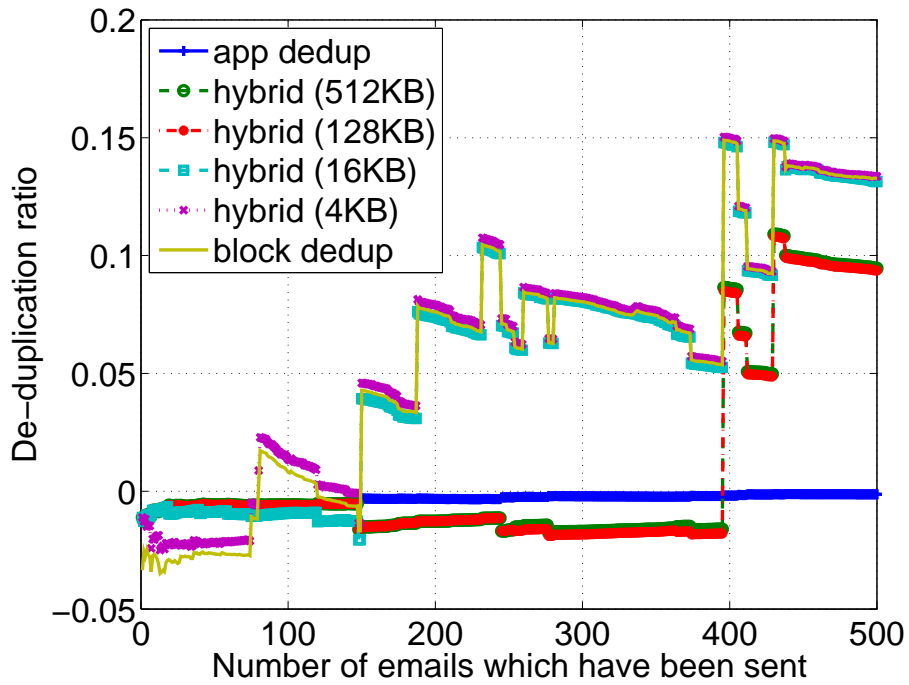


(b) With index

Figure 30: Reduced storage (Enron - corporate dataset)



(a) Without index



(b) With index

Figure 31: Reduced storage (Gmail - single user dataset)

Table 3: Locality of attachments

email id	Date	Size (bytes)	Attachment size (bytes)	Dedup ratio (%)
74	Jan.8	12873	9844	0
75	Jan.8	17805	9844	32.09
80	Jan.8	11957	9844	33.08
81	Jan.8	12012	9844	41.07
86	Jan.9	14896	9593	0

attachments. In Table 3, we show the changes of a relative deduplication ratio compared to the previous email from the 74th email to the 86th email. The first column shows the email ids in the dataset. The second column displays the received date. The third, fourth, and fifth columns show the size of an entire email, the size of an attached file, and relative deduplication ratio compared to the previous email, respectively. Five emails in Table 3 are in the same email thread, where each email has the same subject and title of attachment as the other emails. The 74th email shows 0 deduplication ratio that does not show a deduplication benefit. However, every time the 75th, 80th, and 81st emails were received, we acquire a high deduplication ratio because we do not save the same attachment that was stored in the 74th email already. Interestingly, we see that the 86th email does not show a deduplication ratio, though the title of attachment is the same as other emails. Looking into the attachment, we find that the contents of the attached file in the 86th email have been changed a lot, so even block deduplication cannot find redundancies inside an email. This observation tells us that temporal locality may well be found in emails, and thus, we can exploit the temporal locality with caches.

Table 3 illustrates that 1) file-level deduplication does not detect the same attachments in emails with a different message body. 2) block deduplication can detect the

same attachments, but it has to do chunking of the same attachments every time emails are received, resulting in unnecessary CPU and chunk index overhead. Furthermore, if the email contents are changed heavily, it does not detect redundant parts in the attachments as we see at the 86th email; 3) HEDS can detect the same attachments though a message body is different because it extracts attachments unlike file-level deduplication. Moreover, an attachment in the next email is not chunked if the hash value of the attachment is found, resulting in less CPU and chunk index overhead over block deduplication.

4.11 Memory overhead

In order to find the existence of a chunk in a new email in the existing chunk index, the chunk index is stored in memory. The more chunk indexes means more memory overhead. Here, we evaluate the amount of the chunk indexes produced with different deduplication approaches. The more chunks an email is separated into, the more chunk indexes are created. For our experiments, our system had large enough memory and thus, all the chunk index could be stored in memory. In practice, however, memory would contain only a partial chunk index due to the limit of cache size and handle continuous incoming emails that result in a huge demand in the cache. Then, a cache management scheme, such as LRU, can be used.

As expected, we find that a block-level deduplication shows the largest chunk index overhead, whereas an application-level deduplication shows the least overhead. Figures 32(a) and 32(b) indicate the accumulated chunk index sizes with the Enron dataset

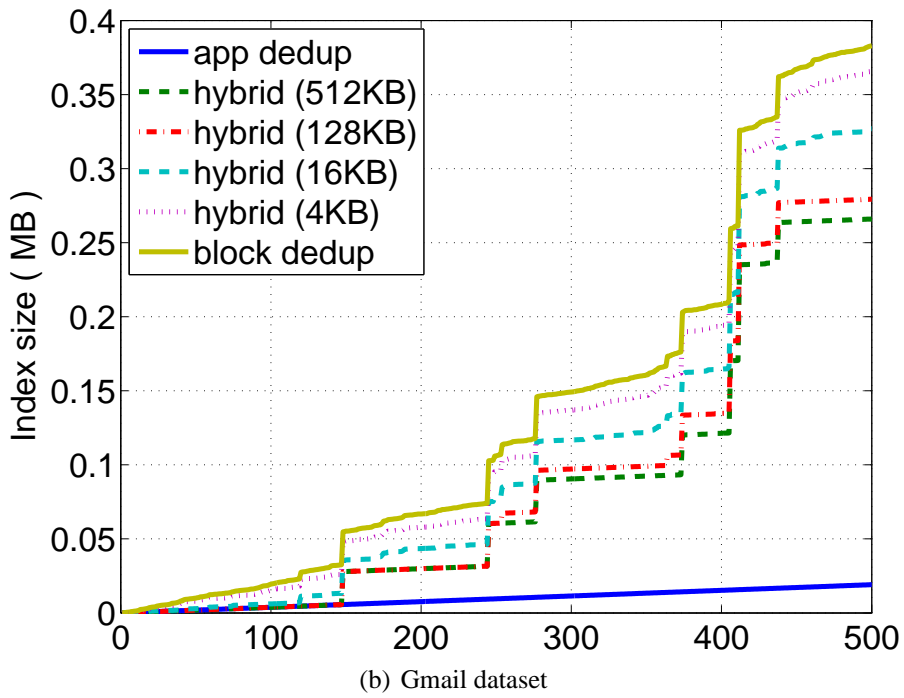
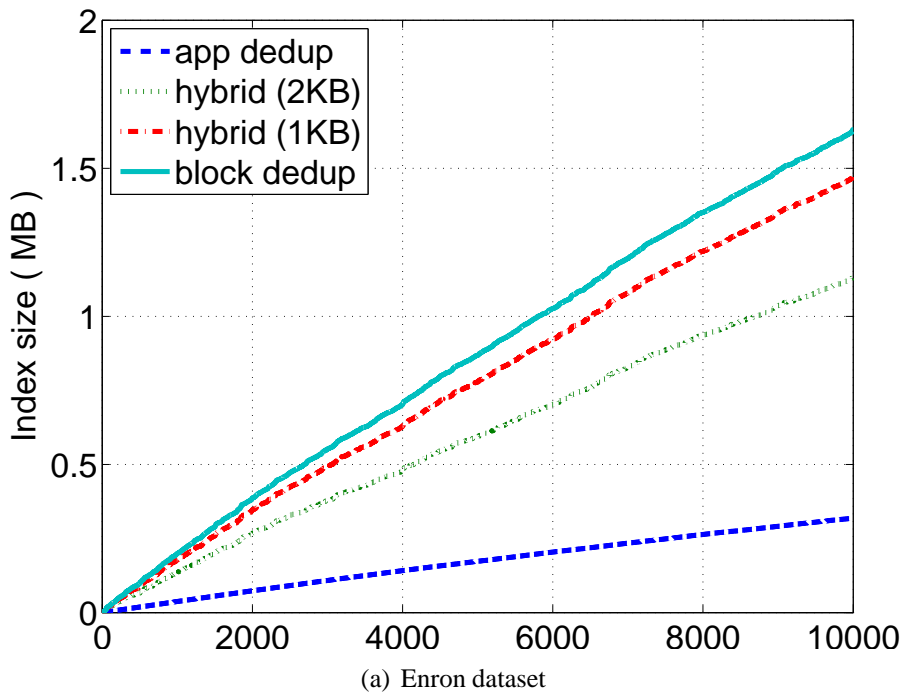


Figure 32: Chunk index overhead

and the gmail dataset, respectively. HEDS shows varying chunk index overhead between application-level and block-level deduplication schemes corresponding to different threshold sizes. HEDS with a small size threshold, e.g. 4 KB or 16 KB, shows a close chunk index overhead than a block-level deduplication. However, HEDS with a large size threshold, e.g. 512 KB or 128 KB, shows a less chunk index overhead than a block-level deduplication. It is observed that there are sudden increases in chunk index overhead over time. This is because many chunks are created from a large sized email that does not have much redundancies compared to previously saved chunks, resulting in creating excessive chunk indexes in memory.

4.12 CPU overhead

Finally, we observe the extra CPU overhead of an application-level, HEDS, and a block-level deduplication scheme. Figure 33 shows the CPU usage measured with the datasets. As for the Enron dataset, an application-level deduplication takes 1.6 times as much CPU as sendmail without a deduplication scheme. A block-level deduplication shows the highest CPU usage, and HEDS's CPU usage stays between the two schemes. The gmail dataset shows the similar relative behavior among the three kinds of schemes, but uses generally higher CPU usage, as it includes attachments and thus a greater number of chunks is generated. In short, HEDS achieves a good tradeoff between an application-level and a block-level deduplication scheme.

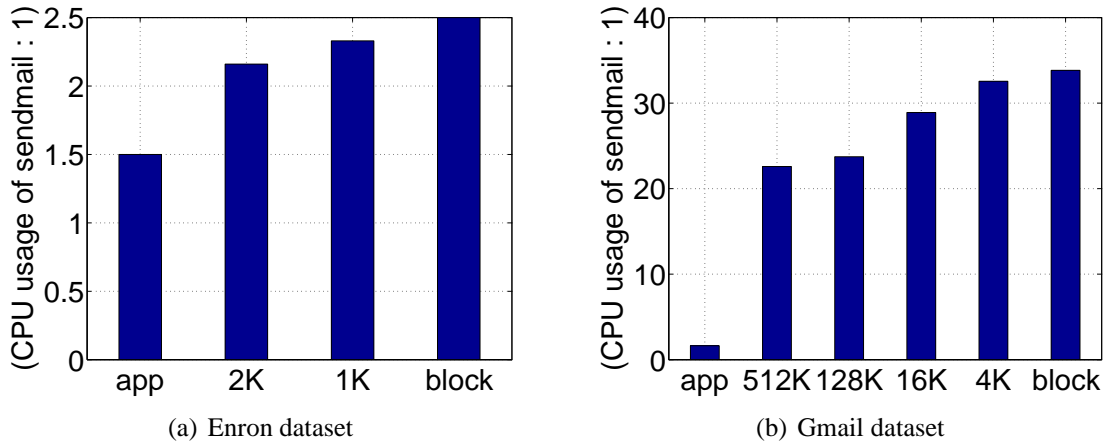


Figure 33: Relative CPU overhead

4.13 Summary

We developed a server-based and hybrid deduplication approach (HEDS) for email servers to minimize data storage size while minimizing the overhead of CPU and memory. It performs hybrid data deduplication adaptively, at the granularity of a file-level or chunk-level, based on the size of emails and the existence of attachments. We have implemented and evaluated the hybrid approach on email servers with real email data sets, and we have shown that it achieves a significantly better reduction ratio of storage consumption than file-level deduplication, and low CPU and memory overheads than variable-size block deduplication.

CHAPTER 5

SAFE: STRUCTURE-AWARE FILE AND EMAIL DEDUPLICATION FOR CLOUD-BASED STORAGE SYSTEMS

In this chapter, we introduce SAFE, a client-based deduplication that is fast and has the same space savings as variable-size block deduplication by using structure-based granularity rather than physical chunk granularity for cloud-based storages.

Cloud-based storages including Dropbox [14], JustCloud [39], and Mozy [52] have been popular as people can access data at any time, anywhere, and with various types of devices such as laptops, tablets, and smart phones. The cloud-based storage services use de-duplication techniques to avoid sending and storing duplicate files (or blocks), reducing network bandwidth and storage space, which gives the subsequent benefit of data upload speed. Existing deduplications (file-level and fixed-size block deduplication) that cloud-based storages use are fast and have a low index overhead, but find fewer redundancies than variable-size block deduplication. However, due to excessive CPU and memory overhead from chunking, indexing, and fragmentation, variable-size block deduplication cannot be used for cloud-based storages.

Thus, we developed SAFE, Structure-Aware File and Email Deduplication, that achieves both fast speeds and shows good space savings in clients by using structure-based granularity for cloud-based storage systems. Evaluation results show that SAFE has as good storage space savings as existing variable-size block deduplication while being as fast as file-level or a large fixed-size block de-duplication.

5.1 Large Redundancies in Cloud Storage Systems

A structure file is a file that consists of meta data and objects like text and image objects. Typical examples of structure files are compressed files (zip, rar), document files (Microsoft Word document, Powerpoint document, and Portable Document Format-PDF), and emails. Everyday, people are creating large numbers of structured files, and cloud-based storages of document suites contain large amounts of structured files. For example, for one of datasets that we used, 89% were structured files and 11% were unstructured files.

We observed that a structure file can be decomposed into various objects with offsets whose positions are dynamically changed based on the location of the objects. As an example, an email is decomposed into multiple objects such as meta data, message body which is text, and attachments. Among attachments, a structure file is further divided into objects like meta data, text, and image objects. Thus, we show that following structures of a file, we can remove redundant objects without expensive chunking.

Based on our observations, we developed SAFE, a fast client-based deduplication that runs on the client-side for cloud-based storage systems. SAFE is as fast in processing time as file level (or fixed size block deduplication) as well as having the same storage space savings as block deduplication by using structure-based granularity.

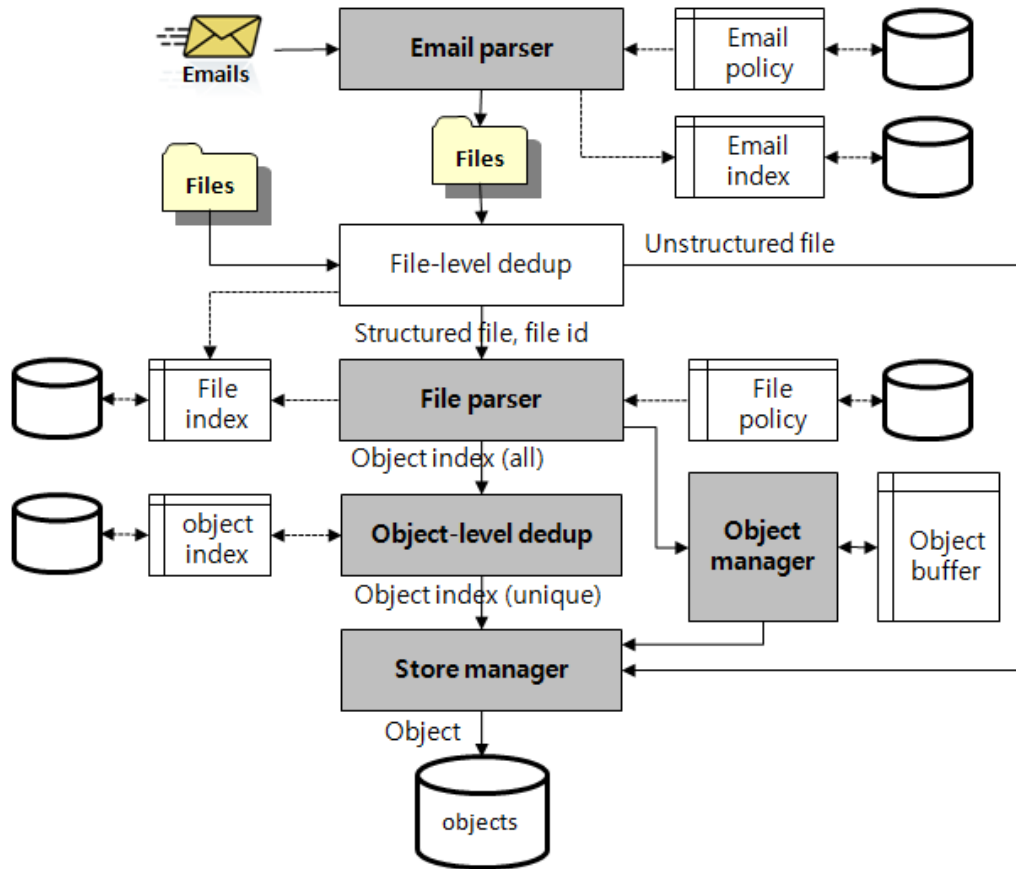


Figure 34: SAFE deduplication architecture

5.2 SAFE Modules

SAFE consists of cooperative modules through which a file is saved into storage. We begin by outlining an architecture of SAFE with modules, and elaborating each module. We explain structures based on which objects are extracted and de-duplicated. Last, we describe how to embed SAFE into a popular cloud storage service, Dropbox.

SAFE incorporates our Structure-Aware de-duplication with existing file-level deduplication. As shown in Figure 34, SAFE system consists of Email parser, File parser,

Object-level dedup, Object manager, and Storage manager modules. SAFE also exploits file-level deduplication to identify redundancy of unstructured files for low CPU and memory overhead. SAFE has two parsers including email parser and file parser. The Email parser module extracts attachments of an email based on email policy, and saves indexes (hash values) of the body and attachments for reconstruction of an email. File-level dedup module receives input data (that is, files) from email parser or file system. If a file is an unstructured file such as an image or a video file, SAFE directly saves the file into a storage after compression at the Store manager module. Otherwise, a structure file is sent to the File parser where a file is decomposed into objects based on a file policy. The File parser sends parsed objects to the Object manager. The Object-level dedup module computes hash value of each object and checks if an object is a duplicate based on the object index table. Last, the Store manager saves unique objects whose indexes are sent from the Object-level dedup into storage after compression. We elaborate each module hereafter.

5.3 Email Parser

The Email parser runs as a light-weight mail filter on a sendmail server [69]. It intercepts an email using Milter [49] APIs when a Mail Transfer Agent (MTA) of a sendmail server receives an email. Milter API is a part of the Sendmail Content Management API that can look up, add, and modify email messages. Figure 35 illustrates how the Email parser works. When an email comes into MTA in an email server, Milter intercepts and sends the email to the Parser that decomposes the email into meta data, body, and

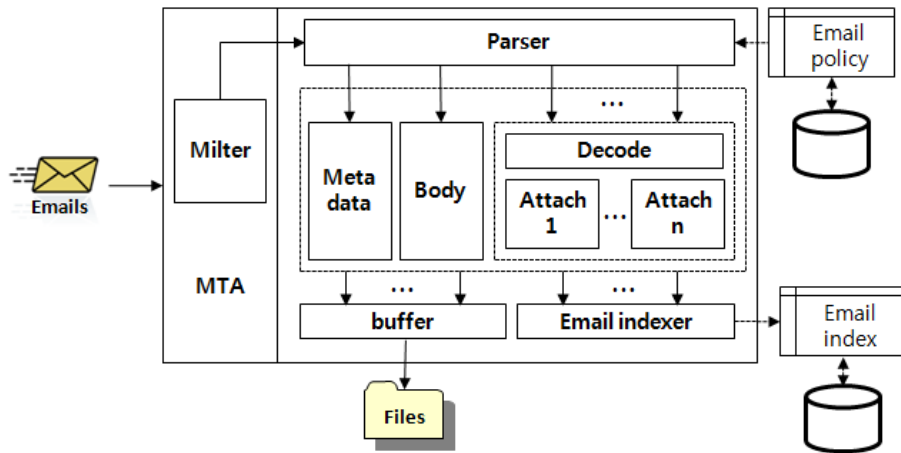


Figure 35: Email parser

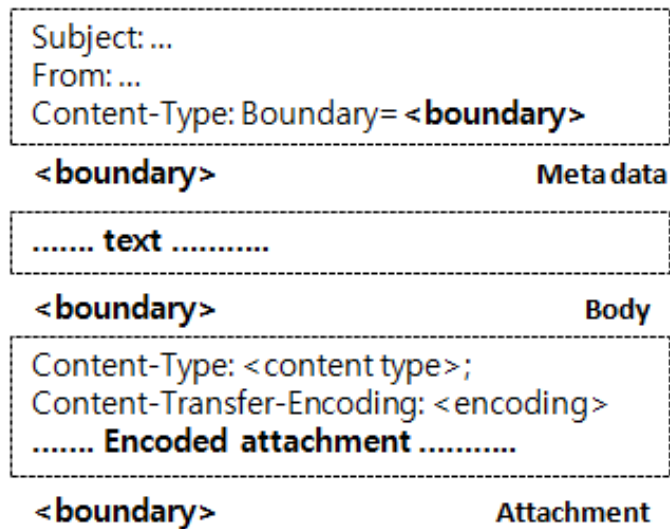


Figure 36: Structure of an email

attachments based on the email policy. The email policy has structure information of an email as shown in Figure 36, and the structure is based on the format of Multipurpose Internet Mail Extensions (MIME) [27]. The Email parser decomposes an email based on a boundary string that are given at “Content-Type:Boundary=” in meta data, which is “<boundary>” in the figure. There can be several attachments that are split by the same boundary string, “<boundary>”. Each attachment may be encoded using different encoded type like “base64” that are designated by “Content-Transfer-Encoding”. Thus, the Email parser runs decoding before processing (sending to other modules) a decomposed attachment.

In Figure 35, the Email indexer computes SHA1 [56] hash values of meta data, body, and attachments decomposed; saves all indexes into an email indexer table by using the unique email ID that is a 14 byte string. The buffer has an array data structure where it holds data decomposed from an email, and sends the array data (that is, files) to the File-level dedup where each file is identified to be unique or redundant.

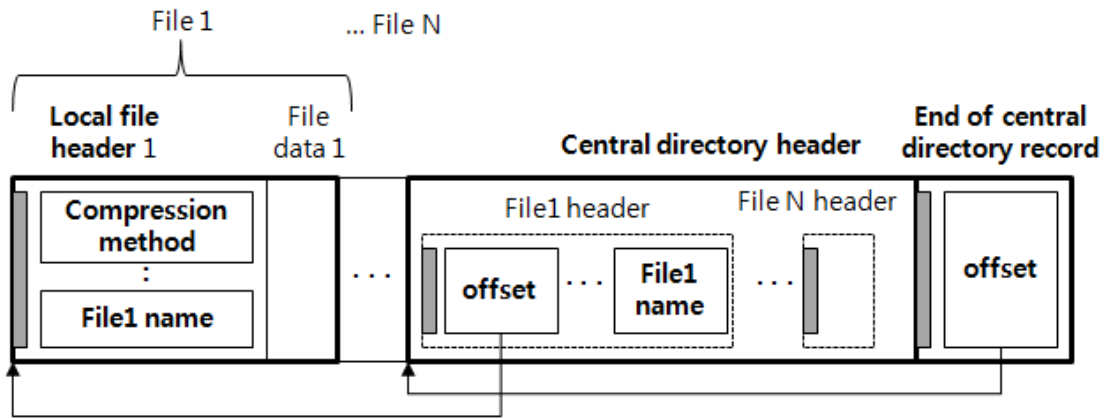
5.4 File Parser

The File parser decomposes three types of structured document files such as Microsoft Word (docx), Powerpoint (pptx), and Adobe Portable Document Format (PDF). SAFE deduplicates a file based on two key aspects: (1) how to extract objects from a file, and (2) what granularity is efficient for deduplication. SAFE uses an object or a combination of objects for a granularity. We explain how the File parser works in detail.

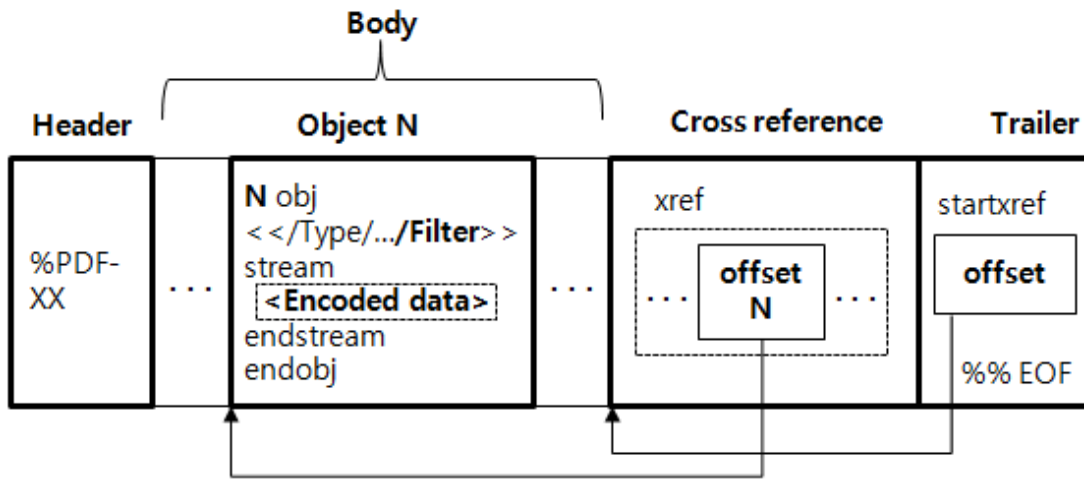
MS word (docx) and powerpoint (pptx) are based on the structure of MS Office

Open XML format which is standardized at ECMA-376 [18] and ISO/IEC 29500 [37], as shown in Figure 37(a). An Open XML file is a ZIP [62] file that contains multiple files such as texts and images. An Open XML file format contains different sections (with meta data and data) that are chained with offsets from the end of a file, and data are tracked down along the offsets upwards. In Open XML format, data are located in sections with local file headers in the beginning of a file. Track to a data starts from “End of central directory record” section with offset of “central directory header” section that describe a directory. The directory has offsets of files in it and through the offsets, a section of a file is accessed. Each file section consists of a “local file header” and file data. The “local file header” contains meta data such as compression method and file name. Likewise, other files are accessed through offsets in the “central directory header” section. In Figure 37(a), Gray bars are signatures. Signatures of “end of central directory record”, “file header” in the central directory header, and the “local file header” are 0x06064b50, 0x02014b50, and 0x04034b50, respectively. The encryption which comes between local file header and file data is not shown.

A PDF physical format contains also multiple sections such as a header, a body, cross references, and a trailer. A PDF structure is defined at ISO 32000 [1], and data are accessed through chains of offsets. The header section is in the beginning of a file, which shows the version of a PDF file. The body section contains objects with a text or an image. The cross reference section has offsets that points to objects in the body section. The trailer section at the end of file has offsets that point to cross reference sections. Thus, data in objects are accessed through offsets from the end of a file upwards. The body



(a) Structure of an MS Office Open XML file



(b) Structure of a PDF file

Figure 37: Physical file format

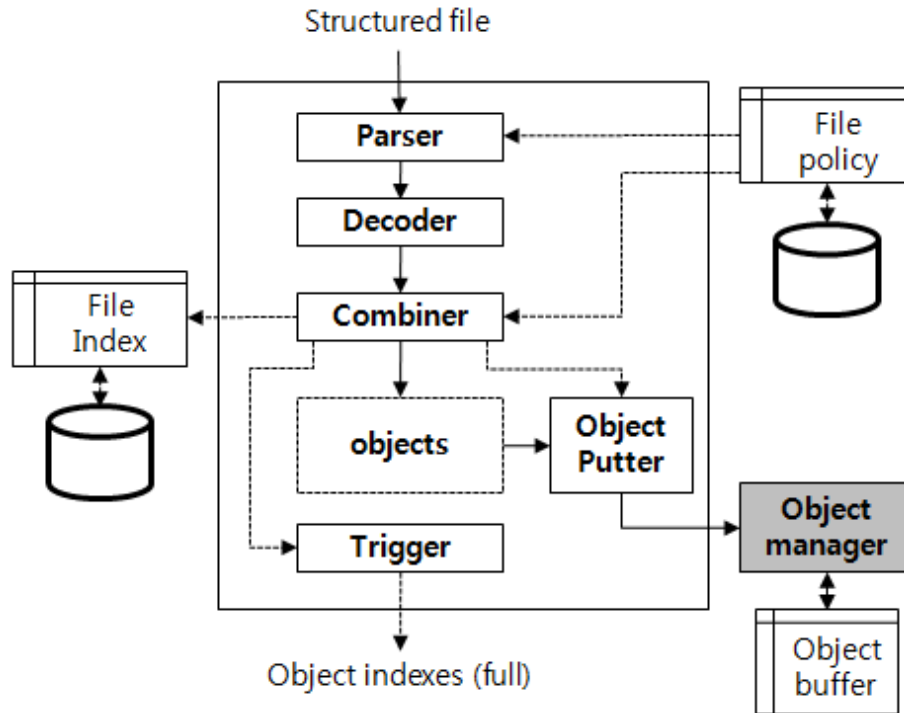


Figure 38: File parser

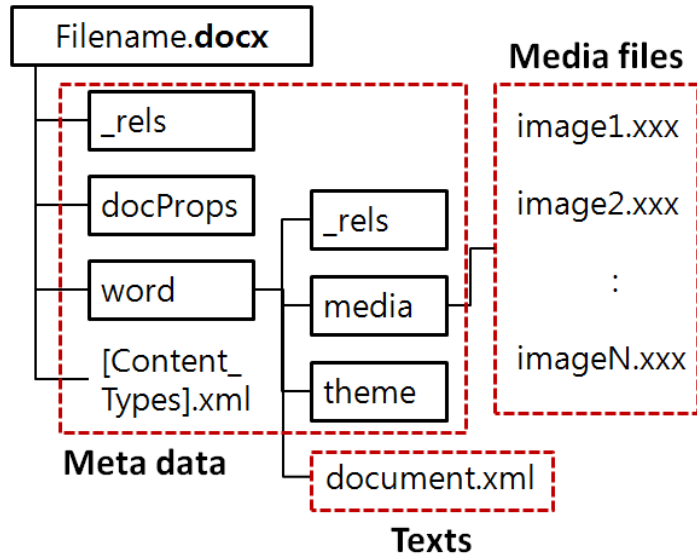
section contains objects surrounded by “obj” and “endobj” each of which may have a text or an image. Two keywords including “stream” and “endstream” surround the data. A stream is encoded by a compression algorithm and can be decoded by the corresponding decompression algorithm shown in the meta data of the object, called ‘dictionary’ (i.e., <</Type../Filter/<decompression algorithm>/..>>). According to ISO 32000, there are 10 different decompression algorithms among which FlateDecode and DCTDecode are used to decode a text stream and a JPEG image stream respectively.

Figure 38 shows how the File parser works. Dotted lines are control flows and solid lines are data flows. Output of the File parser is indexes of all objects including individual objects and combined objects of a file. The File parser receives and parses

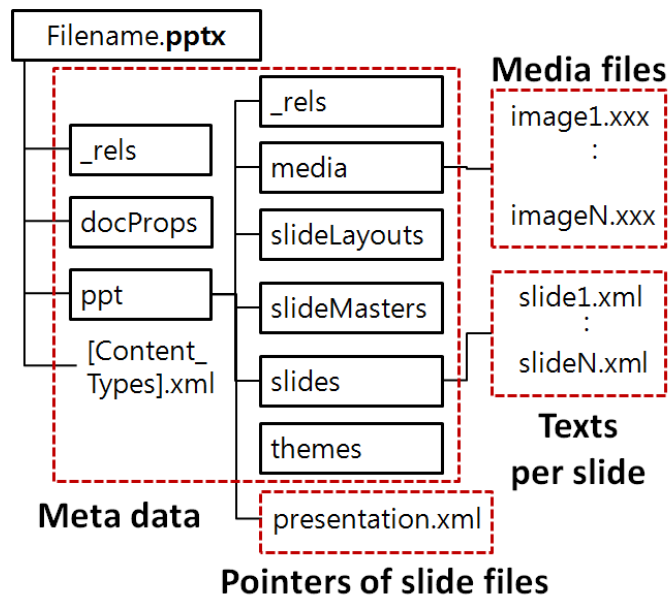
a structured file into objects based on file policy. Encoded objects are decoded based on a decoding algorithm that is specified in structure. The combiner concatenates small objects into a larger object to reduce number of indexes of objects. The combined objects are mainly small-sized meta data whose contents are always changed for each file, in which case we cannot find any redundancy of the objects. A parsed object that is not combined comprises a 5-tuple including hash value of an object, length of an object, ID of container that contains an object (file ID for Open XML format and obj ID for PDF), decoding scheme (if specified), and object itself. A combined object is a concatenation of 5-tuples. The object putter sends an individual object or a combined object into the object manager that subsequently holds the objects in an object buffer until deduplication of the objects is finished. The trigger combines all object indexes and sends them to the object-level dedup where redundancy of objects are identified.

SAFE runs parsing and combining based on a different file policy per file type. To do that, SAFE creates a dynamic instance for each file. SAFE has an abstract base class, FilePolicy, that defines functions to be implemented in derived classes such as DOCXFilePolicy, PPTXFilePolicy, and PDFFilePolicy. The file parser creates a derived class object corresponding to a file type and executes functions of the class object. Thus, a structure file with new format can be implemented as a derived class whose basic functions are already defined.

For combining, SAFE puts together metadata objects that are small, but uses a image and text(content) objects without combination based on logical structures per file



(a) Word (docx)



(b) Powerpoint (pptx)

Figure 39: Logical structure of MS office document file

type. Figure 39 illustrates a logical structure of docx and pptx files. As shown in Figure 39(a), texts of a Word file are contained in a document.xml object, and image objects are under a media directory, and other directories shown in the figure contain metadata objects. Likewise, a Powerpoint file in Figure 39(b) has a media directory, but has different metadata objects. In addition, texts per slide are structured into each individual slide<number>.xml. A presentation.xml holds the pointers of slide objects.

5.5 Object-Level Deduplication and Store Manager

The Object-level deduplication module receives indexes of objects from the File parser, and checks if each index exists in the object index table. If an index does not exist, the index is unique. Unique indexes are saved into an object index table and sent to the Store manager module that fetches objects corresponding to the unique indexes from the Object manager module. If an index does exist in the object index table, the index is redundant. Redundant indexes are excluded for storing. The Object manager module retrieves an object that store manger requests from the Object buffer. The Store manager stores a pair of <object index, object> into object storage after compression. Unstructured files are stored through Store manager without deduplication.

5.6 SAFE in Dropbox

In this section, we describes how SAFE can be embedded into a cloud storage service like Dropbox [14]. Dropbox removes redundancy in network and storage using a large (4MB) fixed-sized block deduplication. Thus, we address how SAFE can improve

data reduction of current Dropbox with minimal additional overhead of processing and memory.

A recent study [13] discovers internal mechanisms of Dropbox by measuring and analyzing packet traces between clients and Dropbox servers. Dropbox is accessed by Web UI (<http://www.dropbox.com>) or dropbox client. We leverage SAFE into a Dropbox client to deduplicate structured files in a client side. Dropbox consists of two type of servers; one is control server and the other is storage server. Control servers hold meta data of files such as hash value of a block and mapping between a file and blocks. Storage servers contain unique blocks in Amazon S3 [3]. Dropbox client synchronizes its own data and indexes with Dropbox servers.

Figure 40 shows how Dropbox works. Circles with numbers show the order in which a file is saved. *File-A* is a file and *Blk-X* is a block which is separated from a file. $h(Blk-X)$ means hash value of a block. Thick $h(Blk-X)$ and ***Blk-X*** are considered as hash values and blocks which already existed before a file is saved. A user's device is mobile phone, tablet, labtop, or desktop. Dropbox follows the next steps to save a file. (1) As soon as a user saves *File-A* into a shared folder in a Dropbox client, fixed-size block deduplication of Dropbox splits a file into blocks based on 4 MB granularity, and computes hashes of objects. If a file is larger than 4 MB, a file is the same as an object, and an hash value of a file is computed. Dropbox uses SHA256 [57] to compute a hash value. (2-4) Dropbox client sends the computed hash values of a file to a control server that returns only unique hash values not found through checking previously saved hash values. In this example, hash of *Blk-B* is returned to a client because hash of *Blk-A* is

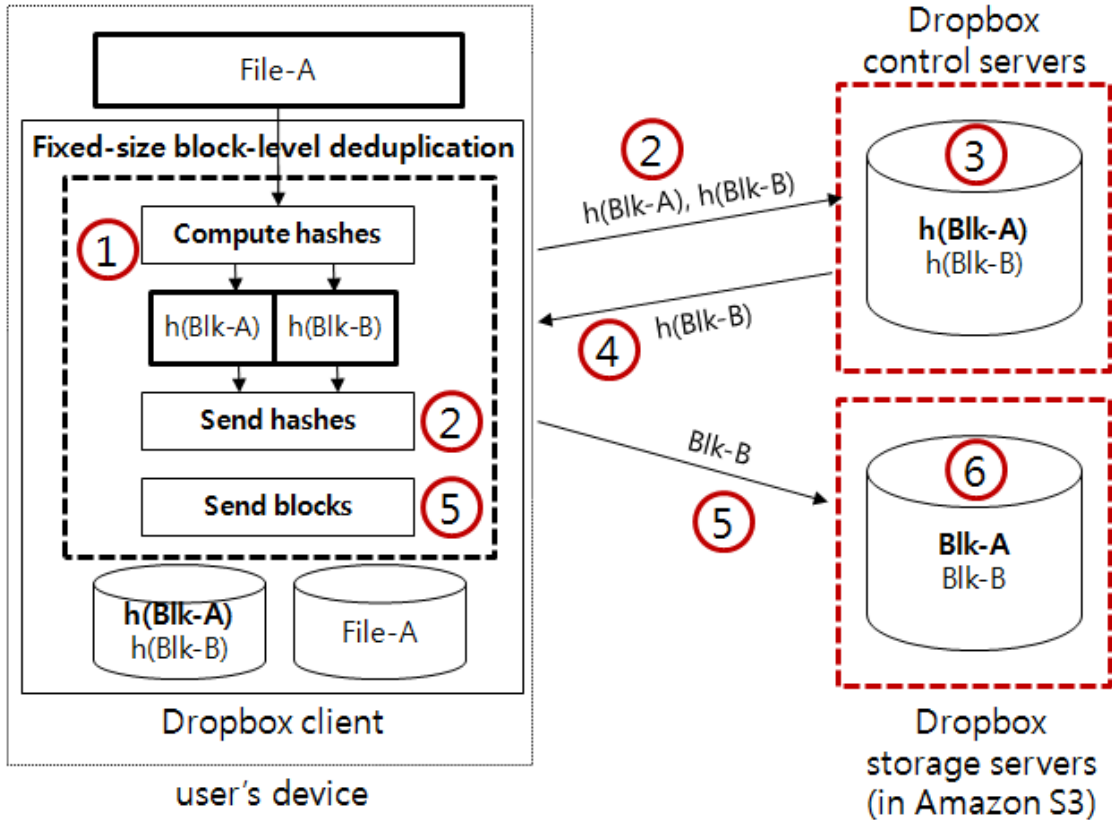


Figure 40: Dropbox internal mechanism

found to be a duplicate. (5-6) A Dropbox client sends to the storage server the blocks of returned indexes. Ultimately, storage servers have unique blocks across all Dropbox clients. Note that storage saving occurs in the server (thanks to not saving *Blk-A* again), and the incurred network cost is reduced thanks to sending *Blk-B* only.

SAFE can complement the fixed-size block deduplication in a Dropbox client as shown in Figure 41. Suppose that an unstructured file (File-A) and a structured file (File-B) are added into a Dropbox folder. The file-level deduplication module checks duplicate files using the file index table whose entry has a pair of <hash value of file contents,

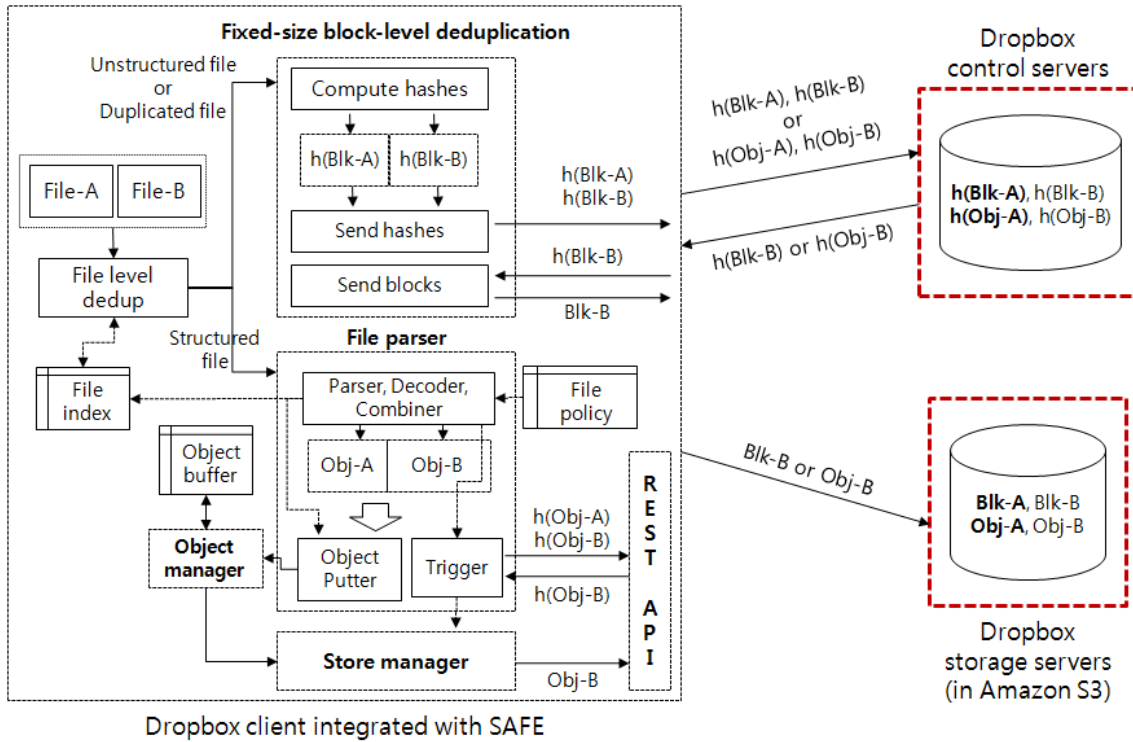


Figure 41: SAFE integration with Dropbox

file path of the first unique file>. For duplicate files, the entry is added into a file index table without savings of a file in local storage. An unstructured file follows the fixed-size block deduplication. A structured file is fed into the File parser, and objects of the file are extracted. The trigger module calls the REST API [15] of Dropbox to send the hash values of objects. The control servers act as an object-level dedup module. We used SHA256 hash function in SAFE for compatibility with Dropbox. The store manager sends objects corresponding to returned hashes from a control server to a storage server through the REST API. Thus, in the integration of SAFE with Dropbox, control servers function as object-level dedup module. In the Figure 41, thick fonts such as $h(\mathbf{Blk-X})$,

$h(Obj-X)$, $Blk-X$, and $Obj-X$ are existent already before $file-A$ and $file-B$ are saved.

5.7 Metrics and Setup

We discuss the performance evaluation criteria and datasets used in this section. We then show the evaluation results of performance and overhead of the proposed SAFE approach, compared with a file-level deduplication that JustCloud [39] and Mozy [52] use, a fixed-size block deduplication that Dropbox [14] uses, and variable-size block deduplication schemes.

The major performance metrics are the deduplication ratio and incurred data traffic amount. The deduplication ratio indicates how much storage space can be saved by removing redundancies, and is computed by Equation (5.1).

$$\left(\frac{InputDataSize - ConsumedStorageSize}{InputDataSize} \right) \times 100 \quad (5.1)$$

Data traffic incurred designates how much data are transferred to a storage that is the amount of unique data out of the input data.

As overhead metrics, we measure the processing time and index size. Since the overhead is proportional to the data size, we compare the processing time and index size overhead relative to the file-level deduplication that has the least overhead.

We collected real datasets of structured files including docx, pptx, and pdf from the file systems and emails of five graduate students in the same department. Table 4 summarizes the information of datasets that were collected from file systems and emails. Individual user’s data is labeled as ‘P-’#, and ‘Group’ is the sum of all personal datasets and ‘no.’ is the number of structured files in each dataset. For the experiments with email

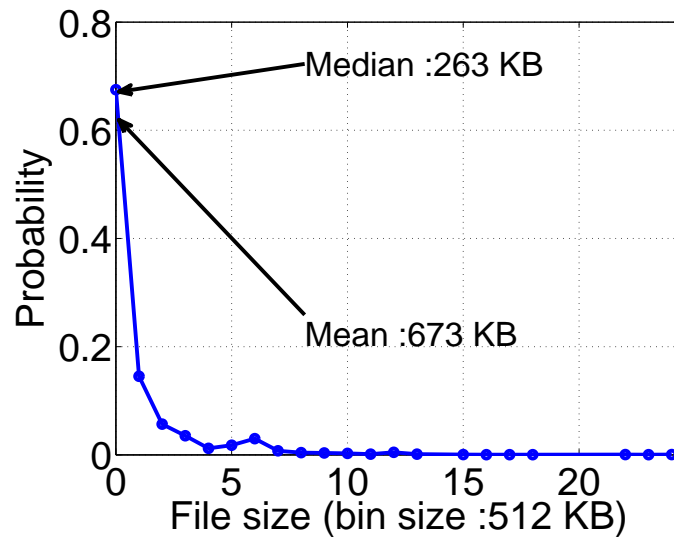
Table 4: Used datasets

Data set	file systems		emails	
	size (MB)	no.	size (MB)	no.
P-1	1,721	4,384	637	955
P-2	509	590	554	720
P-3	266	523	249	480
P-4	869	1,499	358	859
P-5	864	1,430	744	823
Group	4,229	8,426	2,542	3,837

datasets, we deployed two sendmail servers; structured files are attached to emails from a sending sendmail server, and the attached structured files are extracted by the email parser at a receiving sendmail server. Structured files in the file system datasets are fed into the file parser directly.

Figure 42 shows the ranges of the file sizes in the email group dataset whose mean value (673 KB) is relatively small compared to the maximum block size 4 MB of Dropbox. 10 and 20 in x-axis indicate 5 MB and 10 MB, respectively. Meanwhile, we measured the percentages of the structured files among all attached files of five people's emails. As shown in Figure 43, the structured files occupy 89% out of all attached files. PDF occupies 44% and the percentage of docx and pptx is 11%. Image files such as jpg, bmp, and png belong to unstructured file types. Despite the small size of datasets, the high percentage of structured files (89% for all types of structured files and 55% for docx, pptx, and pdf structured files) validates the popularity of structured file types on which SAFE is based.

The datasets used may be considered to be relatively small. However, we note that

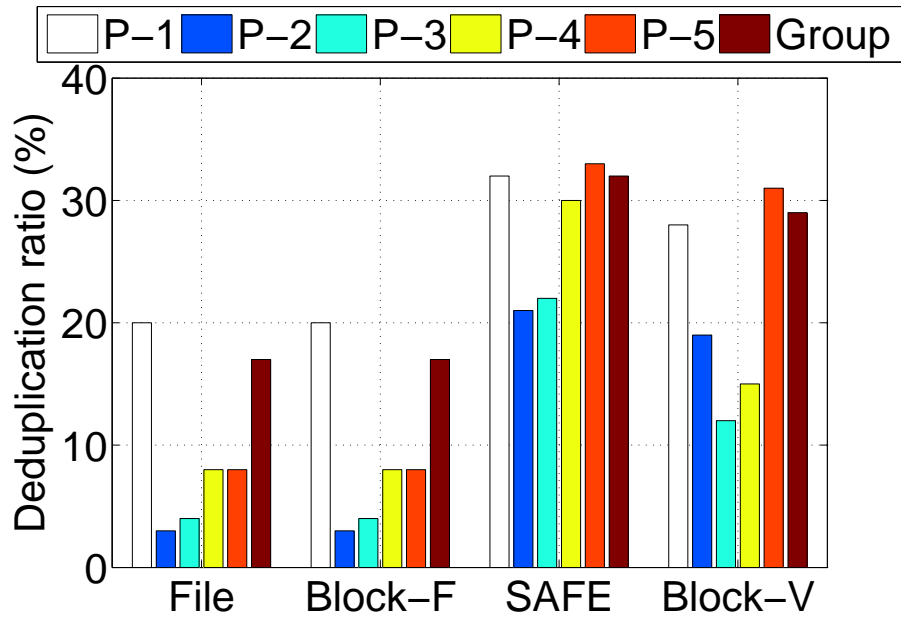


the results obtained in this evaluation will only be stronger if larger datasets of an organization are used, since the redundancy levels would become greater. For the variable-size block deduplication, we use 2 KB, 8 KB, and 64 KB as minimum, average, and maximum chunk sizes, respectively. For fixed-size block deduplication, we use 4 MB as the fixed block size as Dropbox does. Fixed-size block deduplication is thus the same as the file-level deduplication for files smaller than 4 MB. We carried out the evaluations on Fedora 16 Linux operating systems of kernel 2.6.35.9 SMP on Intel Core 2 Duo 3GHz.

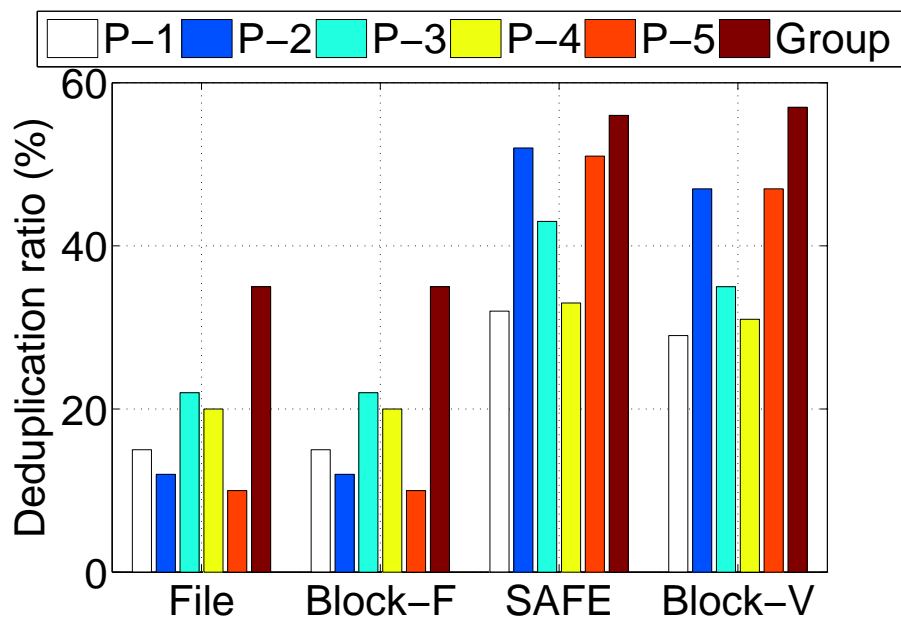
5.8 Storage and Data Traffic Reduction Performance

We first evaluate the deduplication ratio for each dataset. The deduplication ratio of a group is larger than that of each personal dataset. For the file systems, the high deduplication ratio of a group is due to the same or similar content files shared among people in the same department. For emails, the high deduplication ratio of a group is due to duplicates of multiple-recipient emails as well as the same or similar attachments delivered and updated through email threads.

Figure 44 presents the deduplication ratio of six datasets including personal datasets and a group dataset. File, Block-F, and Block-V means file-level deduplication, fixed-size block deduplication and variable-size block deduplication, respectively. Deduplication ratios with the email datasets are higher than those with the file system datasets due to the frequent email threads in addition to shared attached files among people in the same department. Compared to the file-level deduplication in Figure 44 on an average based on group datasets, SAFE can further reduce 15% redundancies and achieves about 40%



(a) File system datasets



(b) Email datasets

Figure 44: Deduplication ratio

better performance than that of the file-level deduplication. For the email datasets, SAFE shows almost 99% of the performance level of the variable-size block deduplication. Furthermore, SAFE's deduplication ratio is better than the variable-size block deduplication in the file system datasets. It is because SAFE can find the boundaries of objects more efficiently in complicated structured files than the variable-size block deduplication, especially for PDF that uses compressions for more individual objects than other structured files such as docx and pptx. Note that file system datasets have twice as many PDF files as email datasets.

We next evaluate the incurred data traffic for group datasets as shown in Figure 45. For file system datasets, SAFE shows the lowest data traffic among all deduplication types: concretely, SAFE has the lowest data traffic with the file system datasets, and the second to the lowest (just behind the variable-size block dedup) with the email datasets. This supports that SAFE can be used as a deduplication technique for personal cloud storage services like Dropbox due to the expected decrease in network bandwidth consumption. In addition, for email datasets SAFE reduces 56% data traffic out of the email group dataset (1.4 GB out of 2.5 GB). Compared to the file-level and fixed-size block deduplications, SAFE has lower data traffic by 30% for the email datasets (and 15% for the file system datasets), which indicates that SAFE efficiently reduces the network bandwidth requirement storing emails to cloud storages.

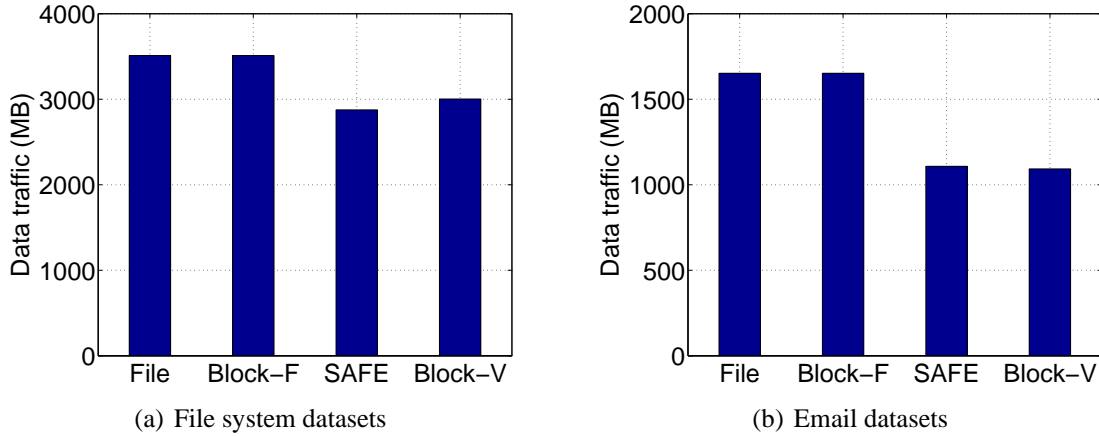
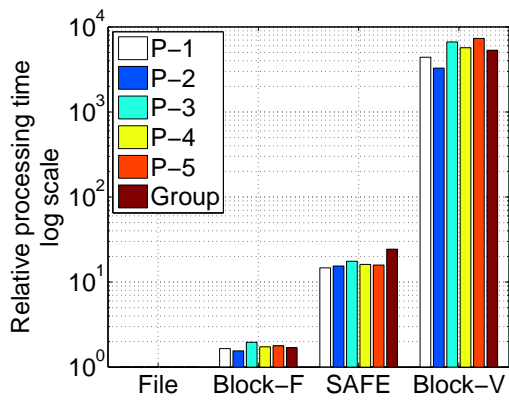


Figure 45: Data traffic incurred (MB)

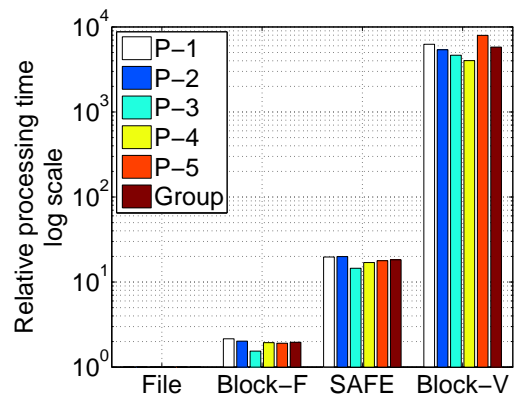
5.9 Memory and CPU Overhead

Here we show the assessments of the processing time and memory overhead. As shown in Figure 46, the file-level deduplication runs the fastest for both datasets types, due to no overhead of separating a file. We present relative processing time based on file-level deduplication that (whose value is 1) is shown as 0 because y-axis is set in log scale. The fixed-size block deduplication shows close processing time overhead to the file-level deduplication. Even if it is slower than the file-level deduplication, SAFE processing is relatively fast on average for the datasets despite that we do not use salient cache management schemes in our implementation. In addition, SAFE is faster by two orders of magnitudes than the variable-size block deduplication.

We now compare the relative index overhead in Figure 47. Like processing time, we present relative index overhead compared to file-level deduplication. SAFE shows 2 to 3 times less index overhead than the variable-size block deduplication. We use a 40

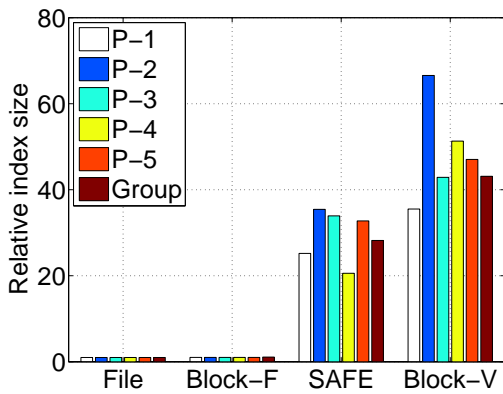


(a) File system datasets

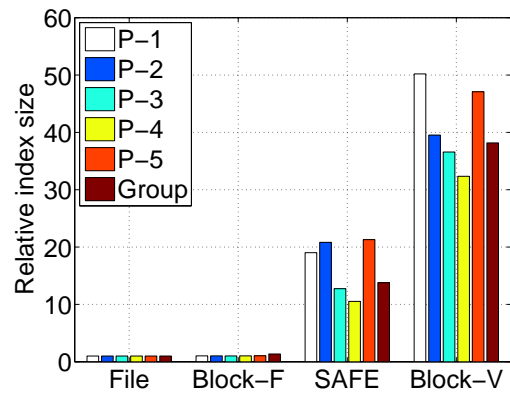


(b) Email datasets

Figure 46: Relative processing time overhead compared to file-level deduplication



(a) File system datasets



(b) Email datasets

Figure 47: Relative index overhead compared to file-level deduplication

bytes-hexadecimal string of SHA1 hash value for a chunk index in all testing deduplication schemes. Though smaller sized chunk index can reduce overhead of variable-size block deduplication, the relative ratios shown in Figure 47 would be maintained. The index overhead increases proportionally to the number of unique chunks. For the email datasets, the numbers of unique chunks for file-level deduplication, fixed-size block deduplication, SAFE, variable-size block deduplication were 2.4K, 2.5K, 33K, and 92K, respectively. For the file system datasets, the numbers for each deduplication scheme were 5K, 5.5K, 155K, and 248K, respectively. SAFE with the file system datasets shows a little more chunk index overhead than with the email datasets. This is because the file system datasets had higher percentages of pdf files than the email datasets. PDF files have a relatively complex structure where files are divided to many small objects, and the current file policy we implemented for PDF saves each object individually without combining. By combining multiple small objects into a large object as in file policy for docx and pptx, SAFE would reduce more chunk index overhead for PDF files.

5.10 Summary

We developed a fast client-based deduplication, SAFE that removes redundant objects based on a structure-based granularity instead of using a physical chunk granularity. Unlike traditional deduplication that is a trade-off between deduplication ratio and processing overhead, SAFE gains benefits of both high deduplication ratio and low processing overhead. Our experiments with real datasets and implementation on a cloud storage client show that SAFE achieves more storage space savings by 10% to 40% and

less data traffic by 20% on average than the file-level and fix-sized block deduplication which are used in existing cloud-based storage services. In addition, SAFE shows permissible processing time on average to be used in a client for cloud-base storage system, and is faster by two orders of magnitude than variable-size block deduplication. Thus, SAFE can be used for deduplication in a client that should be fast and produces low overhead.

CHAPTER 6

SOFTDANCE: SOFTWARE-DEFINED DEDUPLICATION AS A NETWORK AND STORAGE SERVICE

In this chapter, we focus on removing redundancy in a chain between end-systems through a network. As nodes are massively connected through networks, redundancy occurs in various domains (storage and network) and in diverse ways including copying and modifying files, redundant transfers through networks, backup and replication in servers. Simply leveraging data reduction techniques developed in each domain does not give benefits, and even incurs significant redundant processing overhead. In this chapter, we present SoftDance, software-defined deduplication as a network and storage service.

SoftDance chains and virtualizes storage deduplication and network redundancy elimination by using Software Defined Network (SDN) to achieve both storage space and network bandwidth savings while reducing expensive overhead of processing time and memory size. SoftDance uses encoding and indexing schemes for SoftDance middlebox (SDMB) and control mechanisms for an SDN controller. Evaluation results show SoftDance reduces 2-4x more bandwidth than network-wide redundancy elimination technique and achieves equal/close storage space saving to existing the best storage saving techniques.

6.1 Large Redundancies in Network

Redundancies that occur in various domains (storage and network) consume storage spaces and reduce available network bandwidth as nodes are massively connected through networks. In storage domain for data reduction, *data deduplication (Dedup)* has been proposed [7] [40] [41] [44] [46] [63] [78]. Dedup computes indexes of chunks (split from file) and does not store redundant chunks by comparing current indexes with indexes of chunks saved previously. Each index points to a unique chunk. In a network domain, *network redundancy elimination (NRE)* has been studied [4] [5] [71] for data reduction. NRE computes indexes [64] for the incoming packet payload, and removes redundant byte strings in packets by checking packets saved previously. Though Dedup and NRE share the same goal of identifying and removing redundant data, functionalities of the two are orthogonal. Thus, they do not provide any benefits for each other and even incurs redundant processing overhead on both end-systems and networks.

We propose an efficient framework for *software-defined de-duplication as a network and storage service (SoftDance)* to save storage space and network bandwidth, while reducing overhead of processing time and memory overhead. As presented in Figure 48, SoftDance consists of SDMBs (SoftDance middlebox), OpenVSwitches, a SoftDance controller, and lightweight modules at end-systems. SDMB mainly performs encoding and indexing algorithms. SDMB identifies a packet payload for encoding, stores an index of unique packet payload, and replaces redundant packet payload with an index (called encoding). SDMB also maintains an appropriate indexes by communicating with a SoftDance controller. A SoftDance controller provides deduplication function virtualization

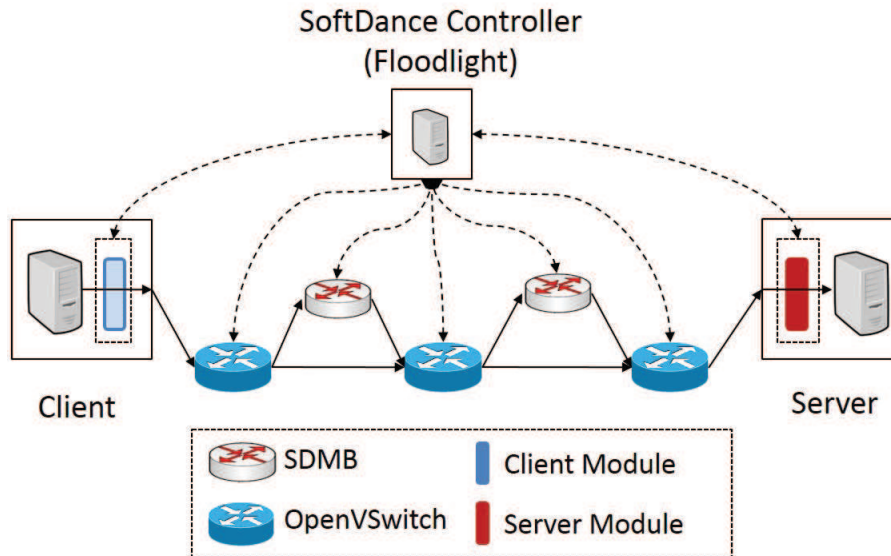


Figure 48: SoftDance architecture

and control mechanisms by coordinating end-systems and network elements. SoftDance uses a packet payload as a unit of comparison, which enables a zero chunking mechanism through deduplication function virtualization. Considering that chunking is a culprit of expensive processing time, SoftDance reduces this processing time significantly. In addition, SoftDance distributes indexes to reduce memory overhead on SDMBs based on hash based sampling [68]. Various index distribution algorithms are designed and implemented in a SoftDance controller.

To validate our approach, we implement the proposed framework and algorithms on both a testbed system and mininet-based emulation by using Software Defined Network (SDN) technologies. We built a testbed system by using OpenVSwitches, a floodlight SDN controller, and Linux based SDMBs that intercept packets using userspace netfilter library. Mininet-based emulation compares SoftDance with Dedup and NRE

techniques based on typical Data Center Network (DCN) topologies including tree, multi-rooted tree, and fat-tree. Our evaluation results from both testbed and mininet-based emulation show that SoftDance reduces 2-4x as much bandwidth as network-wide redundancy elimination (SmartRE) and has equal/close storage space saving to existing storage domain techniques. Furthermore, in scenarios of both end-systems and networks performing deduplication redundantly, SoftDance achieves much efficient processing and memory overhead.

The rest of the chapter is organized as follows. We begin by explaining Software Defined Network as a background information in Section 6.2. We describe the design and implementation issues on packet encoding and indexing algorithms of SDMB and a system coordination scheme of a SoftDance controller in Section 6.3. We evaluate our approach in Section 6.7, and Section 6.12 concludes this chapter.

6.2 Software Defined Network

SoftDance is based on Software Defined Network (SDN) to set up efficient paths for removing more redundancies and reducing indexes in networks. Software Defined Network is a new paradigm that separates the control plane that computes forwarding rules and the data plane that forwards data packets in a network element. As shown in Figure 49, SDN moves the control plane from a switch to a centralized SDN controller that has global network view and decides paths based on application requirements or policies. When a data packet arrives at a switch without a corresponding forwarding rule, the switch asks a controller. Then, the controller sets forwarding rules to the switch, and

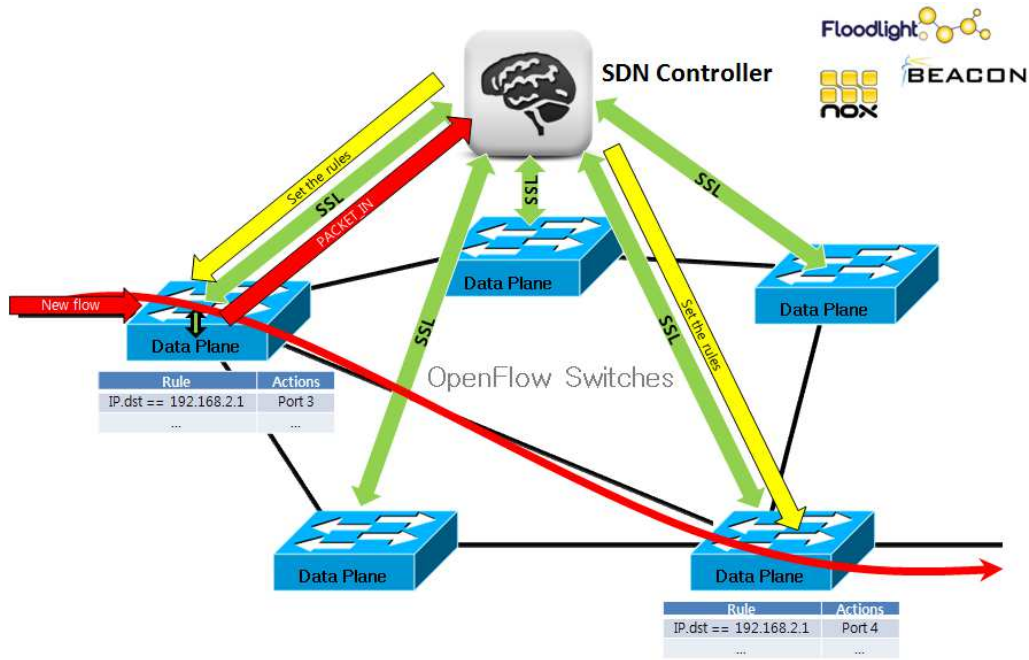


Figure 49: Software defined network

data packets are forwarded based on the rules.

6.3 Control and Data Flow

To effectively reduce redundancies in chains from clients to servers through a network, SoftDance coordinates clients, servers, and SDMBs using a SDN controller. A SDN controller with global network view controls data transfer service requests of clients, and provides an efficient path from a client to a targeted server, leading to low processing and memory overhead. Next, we explain the design and implementation of SoftDance. We start by presenting control and data flows. We then elaborate on the encoding scheme processed at SDMBs. Last, we describe four distributed hash indexing algorithms.

SoftDance uses control flows to set up a service request from a client and data

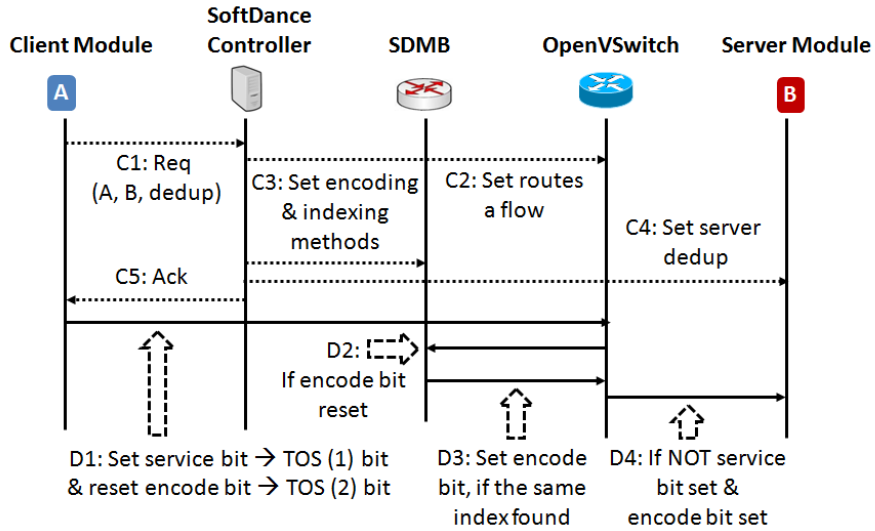


Figure 50: SoftDance control and data flows

packets flow based on the set-up through switches and SDMBs. A SoftDance controller coordinates control flows through communication with clients, servers, SDMBs, and OpenVSwitches.

As drawn in Figure 50, the SoftDance process starts with a client's deduplication service request (C1). A client sends the request along with the client's and server's IP addresses to a SoftDance controller. When a SoftDance controller receives the service request from a client, the controller performs Algorithm 4. A controller computes and selects a path between a requested client and a targeted server, retrieves SDMBs and switches on the selected path, and computes hash ranges of retrieved SDMBs. Then, a controller pushes flow table entries into switches on the path (C2). Figure 51 illustrates forwarding tables with entries in each switch. A controller sends hash ranges computed to SDMBs on the path, and SDMBs set up the hash range for each path (C3). Then, a

Algorithm 4 SoftDance Controller

Input: inPacket(ServiceRequest)**Output:** outPacket

```
1: senderIP = getSenderIP(inPacket)
2: // set up service
3: srcIP = getSrcIP(inPacket)
4: dstIP = getDstIP(inPacket)
5: (SDMBList, switchList) ← setupPath(srcIP, dstIP)
6: computeHashRange(SDMBList)
7: pushFlowEntry(switchList, SDMBList)
8: assignHashRange(SDMBList)
9: registerToService(srcIP, dstIP)
10: outPacket ← “confirm”
11: forward outPacket(senderIP)
```

controller sends a configuration message to a destination node for preparing deduplication in the storage system (C4). Finally, a controller registers a service with a pair (A,B) and acknowledges to the requesting client.

When a client’s SoftDance request has been approved, a client starts sending data packets. For forwarding data packet, we use most significant two bits on TOS fields. The first bit on TOS is called a service bit that represents if a data packet uses our deduplication service. The second bit on TOS is called a encoding bit that indicates if a data packet has been encoded by the previous SDMB. A client sends a data packet after setting a service bit and resetting an encoded bit (D1). When a switch receives a data packet, if a service bit is on and an encoded bit is off, a switch forwards the data packet to a SDMB (D2). Otherwise, the data packet is forwarded to the next switch. For example, as shown in Figure 51, if the service bit is on and the encoded bit is off (as shown at the third row), switch 1 forwards the data packet to a SDMB through port 3. Otherwise, data packets

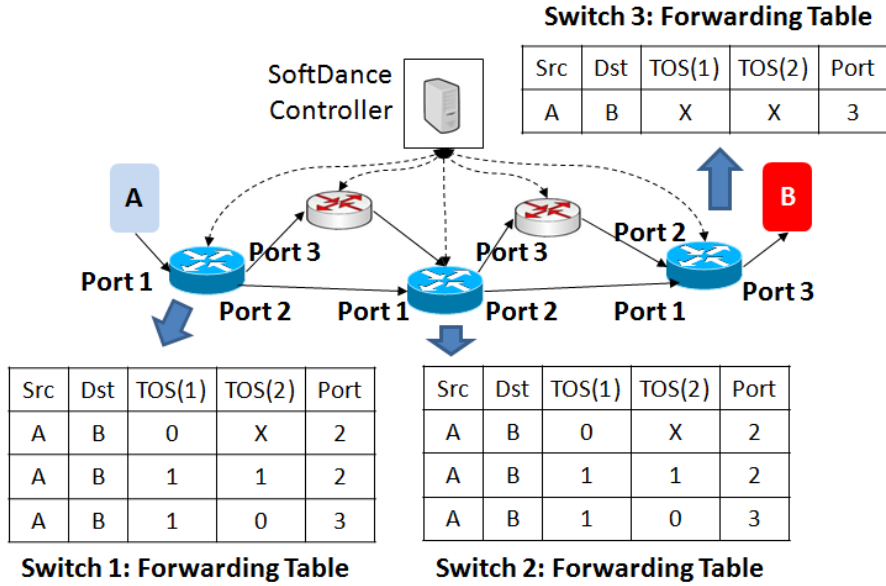


Figure 51: SoftDance forwarding table example

are sent to the next switch (switch 2) through port 3. A SDMB checks redundancy of a data packet while comparing an index of its payload with previously saved indexes. If the same index exists, the data packet is redundant. In this case, a payload is replaced with an index and an encoded bit is set (D3). When a server receives a data packet, if an encoded bit is not set, an index and data itself is saved. Otherwise, only index is stored for the future data reconstruction (D4).

6.4 Encoding Algorithms in Middlebox (SDMB)

An SDMB takes SoftDance service packets that are forwarded by a switch, and encodes redundant packets among the taken packets. In this section, we explain how to take packets and encode redundant packets.

Algorithm 5 explains packet processing in SDMB. SDMB computes a path ID of

Algorithm 5 Packet Processing in SDMB

Input: inPacket**Output:** outPacket

```
1: // pathID is <srcIP> - <dstIP>
2: pathID = getPathID(inPacket)
3: payload = getPayload(inPacket)
4: hashKey = computeHash(payload)
5: hashRangeKey = computeHashRangeKey(hashKey)
6: if hashRangeKey  $\in$  hashRange(pathID) then
7:   if hashKey  $\in$  indexTable then
8:     // redundant packet - encode
9:     replacePayload(hashKey, inPacket, outPacket)
10:    recomputeChecksum(outPacket)
11:   else
12:     // unique packet
13:     saveToIndexTable(hashKey)
14:     outPacket  $\leftarrow$  inPacket
15:   end if
16: else
17:   outPacket  $\leftarrow$  inPacket
18: end if
19: forward(outPacket)
```

a packet based on a source IP address and a destination IP address from the packet header, and retrieves the payload from the packet. Then, SDMB computes a hash range key of a packet by using SHA1 hash key [56]. Though it is implementation-specific, we use SHA1 hash key for uniform distribution of dataset. To compute hash range key, we take the 18 most significant bits from a SHA1 hash, use a modulo operation with 100, and divide the remainder by 100 to have a range of floating point from 0 to 1. If the computed hash range key is in the hash range that is set by a controller during set-up phase (we defer how a controller compute hash range for a SDMB in next section), SDMB compares the

hash key of a packet with hash keys (indexes) saved previously. When current hash key exists in index table, the current packet is redundant. Payload of a packet is replaced by a hash key and checksum is recomputed for continuous forwarding. If a packet is unique, a hash key is saved into index table for future comparison, and a packet is sent to next hop without encoding the index.

We implement an SDMB as a userspace program that is a callback function based on `libnetfilter_queue` userspace library [60]. An SDMB runs on a Linux bridge that connects the incoming and outgoing network interfaces. To intercept an incoming packet, we set up the iptables rules in a filter table. We set up iptables rules with OUTPUT for a client module, a FORWARD for an SDMB, and INPUT for a server module along with iptables-extension NFQUEUE [59]. Whenever packets come in, packets are given to a userspace program through netfilter queue. A userspace program handles an incoming packet and a processed packet is forwarded back to either a network elements such as switches and SDMBs or a server.

6.5 Index Distribution Algorithms

An SDMB stores indexes of unique packets to compare redundancy of future packets. As the large amount of indexes cause significant processing and memory overhead, we propose distributed indexing mechanism. By using hash-based sampling [68], a SoftDance controller distributes hash ranges to SDMBs on a route of a flow, and each SDMB handles only a data packet whose hash range key belongs to a hash range assigned by a controller. In this manner, SoftDance can reduce processing time and memory size by

Algorithm 6 Compute hash range (uniform, merge)

Input: sdMList, pathList**Output:** nodes with hash range

```
1: for all path  $\in$  pathList do ▷ retrieve each path
2:   if approach == “uniform” then
3:     fraction = 1 / numSDMBs(path)
4:   else if approach == “merge” then
5:     totalDegree = getTotalDegree(path)
6:   end if
7:   sdMBs = getSDMBs(path)
8:   range = 0
9:   for all sdMB  $\in$  sdMBs do ▷ a sdMB in a path
10:    if approach == “merge” then
11:      fraction = sdMB.getDegree() / totalDegree
12:    end if
13:    sdMB.lowerBound = range
14:    sdMB.upperBound = range + fraction
15:    range = range + fraction
16:  end for
17: end for
```

handling a data packet only once on a flow. In this section, we describe four different index distribution algorithms.

SoftDANCE-full (SD-full): SD-full is an approach with a full hash ranges(0,1). Thus, using SD-full, an SDMB processes all incoming data packets and holds indexes of the unique packet among incoming packets. The index size complexity per route is $O(n*m)$ where n and m are the number of unique packets and the number of SDMBs on a path respectively.

SoftDANCE-uniform (SD-uniform): ND-uniform distributes hash ranges uniformly to all SDMBs over a flow path. Each SDMB handles only packets whose hash-range key is

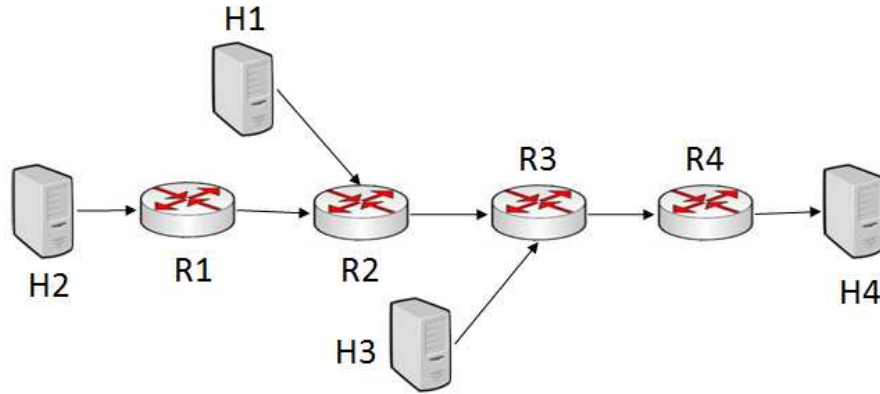


Figure 52: A network topology with three routes

in its hash-range. This scheme reduces index sizes compared to ND-full with the trade-off of reducing bandwidth saving. As presented in Algorithm 6, a SoftDance controller retrieves a path between a client and a server, and computes uniform fractions of SDMBs on a path. Then, a controller assigns a *disjoint hash range* of each SDMB by accumulating sequentially from the closest SDMB (to client) to the farthest SDMB. For example, in Figure 52, a path $H1-H4$ has three SDMBs. Thus, each SDMB has an fraction of $\frac{1}{3}$ or 0.33. Hash ranges are computed from the first SDMB ($R2$) to the last SDMB ($R4$) on a path, starting from 0 by accumulating the hash ranges: $R2$ is assigned $[0,0.33)$, where 0 is inclusive and 0.33 is exclusive. In this manner, $R3$ and $R4$ are assigned $[0.33, 0.66)$ and $[0.66, 1)$ respectively.

Table 5 demonstrates an example how many memory indexes SD-uniform produces on a topology of Figure 52. ‘-’ means a SDMB is not on a path. ‘0 (dup A)’ means a packet A is redundant and none of redundant index is stored. Suppose all clients send two packets A and B , and hash range keys of packets A and B are 0.3 and 0.7 respectively.

Table 5: SD-uniform hash ranges and generated index size

	Path	R1	R2	R3	R4
Hash range	H1-H4	-	[0,0.33)	[0.33,0.66)	[0.66,1)
	H2-H4	[0,0.25)	[0.25,0.5)	[0.5,0.75)	[0.75,1)
	H3-H4	-	-	[0,0.5)	[0.5,1)
Index	H1-H4	-	1 (A)	0	1 (B)
	H2-H4	0	0 (dup A)	1 (B)	0
	H3-H4	-	-	1 (A)	0 (dup B)
	Total	0	1	2	1

Assuming clients $H1$, $H2$, and $H3$ send sequentially, total memory size to be stored is 4. Concretely, hash ranges are computed uniformly among SDMBs on a path. When a client $H1$ sends packets A and B , SDMBs $R2$ and $R4$ stores indexes of A and B . An SDMB $R3$ just forwards data packets without storing indexes because hash range keys of A (0.3) and B (0.7) are not in a hash range $[0.33,0.66)$ of $R3$. When a client $H2$ sends packets A and B , SDMB $R2$ finds a data packet A is redundant, and SDMB $R3$ stores an index of data packet B whose hash range key (0.7) is within hash range $[0.5,0.75)$ of SDMB $R3$. Likewise, a data packet B from a client $H3$ is found to be redundant at SDMB $R4$, not storing an index again. In this manner, SD-uniform reduces index sizes from 8 (in case each SDMB stores indexes of packet A and B) to 4. The complexity of index size per path is $O(n)$, where n is the number of unique packets on a flow path.

SoftDANCE-merge (SD-merge): SD-merge assigns the disjoint hash ranges only for the SDMBs that have more than one incoming flows of the same destination (merge). As presented in Algorithm 6, SD-merge counts total incoming degree of merge SDMBs on a path. Then, a fraction of an SDMB is computed by $\frac{\text{incoming degree of a SDMB}}{\text{total incoming degree of merge SDMBs on a path}}$. `sdMB.getDegree()` function returns 0 if a SDMB is not merge node, leading fraction to 0.

Table 6: SD-merge hash ranges and generated index size

	Path	R1	R2 (merge)	R3 (merge)	R4
Hash range	H1-H4	-	[0,0.5)	[0.5,1)	-
	H2-H4	-	[0,0.5)	[0.5,1)	-
	H3-H4	-	-	[0,1)	-
Index	H1-H4	-	1 (A)	1 (B)	-
	H2-H4	-	0 (dup A)	0 (dup B)	-
	H3-H4	-	-	1 (A), 0 (dup B)	-
	Total	0	1	2	0

Hash ranges are computed by accumulating fractions, starting from 0 like SD-uniform. In Figure 52, for a path H1-H4, there are two merge SDMBs, R2 and R3. Thus, R2 and R3 are assigned [0,0.5) and [0.5,1) respectively, but R4 is not assigned hash ranges; that is, incoming packets to R2 are just forwarded to the next hop (in this case, H4) without encoding.

Table 6 shows how many indexes SD-merge produces. Paths H1-H4 and H2-H4 have two merge SDMBs (R2 and R3), and H3-H4 has only one merge SDMB (R3). When a client H1 sends data packets A and B, R2 stores index of a packet A and R3 stores index of a packet B. Data packets sent from H2 are found to be redundant at R2 and R3. When a client H3 sends data packets, an index of data packet A is stored at R3 but an index of data packet B is found to be redundant. R1 and R4 just forwards packets because their hash ranges are out of [0,1). The total index size of SD-merge is now 3 that is lower than that of SD-uniform. This shows that assigning hash ranges to only merge nodes can find more redundant packets.

SoftDANCE-optimize (SD-opt): As both SD-uniform and SD-merge assign the hash range based on the flow path information, they may not be able to consider the dynamic

Algorithm 7 runOptHashRange: Compute hash range (optimize)

Input: sdMBList, pathList**Output:** nodes with hash range

```
1: rProfileList  $\leftarrow$  importRProfile()  $\triangleright$  matchp,q, matchSizep,q
2: pathList(packetp)  $\leftarrow$  importPacketCounts()
3: pathList(packetp,unique)  $\leftarrow$  importUniqueCounts()
4: // hash range is set to lowerbound, upperbound in a node
5: solveLP(sdMBList, pathList, rProfileList)  $\triangleright$  by LP
6: range = 0
7: for all path  $\in$  pathList do
8:   for all sdMB  $\in$  path do  $\triangleright$  a sdMB in a path
9:     fraction = sdMB.fraction()  $\triangleright$  set by solveLP()
10:    setHashRange(sdMB, range, range+fraction)
11:    range = range + fraction
12:   end for
13: end for
```

conditions such as network traffic, packet redundancy, and resource constraints. To get better hash ranges, we use a linear programming (LP) formation of SmartRE [5] for SD-opt scheme. Algorithm 7 presents how to compute hash ranges based on LP. To run LP, SD-opt needs input constants: (1) redundancy profile that indicates how many packets across paths are redundant (denoted as $\text{match}_{p,q}$), and how many bytes across paths are redundant (denoted as $\text{matchSize}_{p,q}$). (2) number of packets that passed SDMBs on a path p (denoted as packet_p). (3) number of unique packets that passed SDMBs on a path p (denoted as $\text{packet}_{p,\text{unique}}$). SDMBs maintain these input constants and a SoftDance controller uses the input constants that are collected from SDMBs. The $\text{solveLP}()$ function runs LP, computes fractions of SDMBs which are results of LP, and sets the fractions into SDMBs. The $\text{setHashRange}()$ function computes hash range with fractions of SDMBs on a path; concretely, lower bound and upper bound are computed for each hash range.

Adopted formulation of SD-opt is different from the formulation of SmartRE [5]. SoftDance stores only indexes while SmartRE stores packet as well as indexes, which changes the memory constraint in our formulation. Also, SoftDance performs indexing and encoding at SDMBs, but SmartRE runs storing packets and decoding encoded packets, which changes the processing constraints in our formulation. Formulation of SD-opt has three constraints: memory constraints, processing constraints, and fraction constraints. For memory constraint in Equation 6.1, each SDMB stores all indexes of unique packets that are within hash ranges assigned, and the index sizes in the SDMB should be less than available memory size. $d_{p,r}$ is a fraction of a packet that an SDMB r can hold on a path p . $indexSize$ is 40 byte of a hash key string. $packet_{p,unique}$ is the number of unique packets on path p . M_r is the maximum available memory of a SDMB r .

$$\forall r, \sum_{p:r \in p} d_{p,r} \times packet_{p,unique} \times indexSize \leq M_r \quad (6.1)$$

For the processing constraint in Equation 6.2, each SDMB checks hash range and index table for checking redundancy, and encodes redundant packets. The total processing should be less than maximum available processing capability, L_r . $packet_p$ is the number of packets passing a SDMB on a path p . $match_{p,q}$ is the number of packets matched across paths p and q .

$$\forall r, \sum_{p:r \in p} d_{p,r} packet_p + \sum_{p,q:r \in p,q} d_{q,r} match_{p,q} \leq L_r \quad (6.2)$$

Table 7: REST API URIs

URI	Method	Description
/wm/re/cmd/<op>/<ip>	GET	get hash range (ip: ip address of sdMB)
/wm/re/cmd/hashRange/<op>	GET	signal to compute hash range
/wm/re/set/<op>	POST	send node information(ip, mac addr) to controller
/wm/re/get/<op>	GET	retrieve paths

$$\forall p, \sum_{r:r \in p} d_{p,r} \leq 1 \quad (6.3)$$

The third constraint is shown in Equation 6.3 where the maximum sum of the fraction on a path is 1. The objective shown in Equation 6.4 is to find the largest amount of redundant packets considering the storage space and bandwidth savings. $matchSize_{p,q}$ is the total size of matched packets across paths p and q . Our objective is different from SmartRE that is to achieve only bandwidth savings.

$$max \left(\sum_p \sum_r \sum_{q:r \in q} d_{q,r} \times matchSize_{p,q} \right) \quad (6.4)$$

6.6 Implementation: REST, JSON, Middlebox

We use Floodlight [26] to implement a SoftDance controller. We implement a Floodlight module [25] that computes hash ranges. A client module, SDMBs, and a server module communicate with a SoftDance controller through REST API using cURL [73]. We add REST API URIs into Floodlight module for communication. SDMBs

```

"hashRanges": [
  {
    "hashRangeLB": "0.5",
    "hashRangeUB": "1.0",
    "path": "192.168.2.12_192.168.2.5"
  },
  {
    "hashRangeLB": "0.5",
    "hashRangeUB": "1.0",
    "path": "192.168.2.11_192.168.2.5"
  }
]

```

Figure 53: JSON format example: response of hash range URI

use C++ JSON parser [38] to parse JSON data (with hash ranges) that is delivered from a SoftDance controller. A few important URIs are shown in Table 7. For example, “/wm/re/set/<op>” is used for a SDMB to send node information including IP and MAC addresses to a controller using POST method. <op> is one of “uniform”, “merge”, and “opt”. “/wm/re/cmd/<op>/<ip>” is used for a SDMB to receive hash range from a controller.

Figure 53 describes JSON format of an example responded by the “get hash range” URI. SD-opt requires input constants including $packet_{p,unique}$, $packet_p$, $match_{p,q}$, $matchSize_{p,q}$, M_r , L_r to run a LP. In our prototype, SDMBs maintain the input constants during each SoftDance service.

6.7 Experiment and Emulation Setup

We measure the performance and overhead of SoftDance compared to other existing storage and network data reduction techniques. We begin by describing our setup

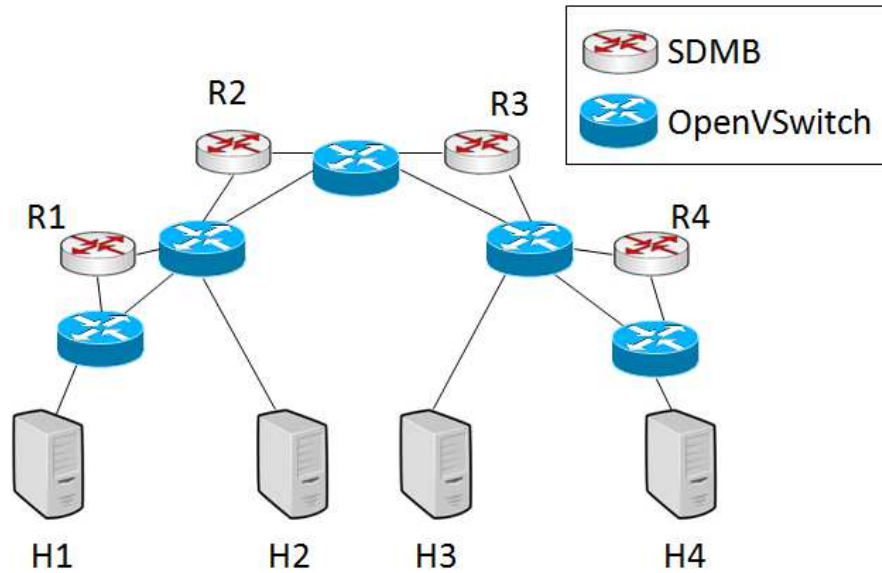
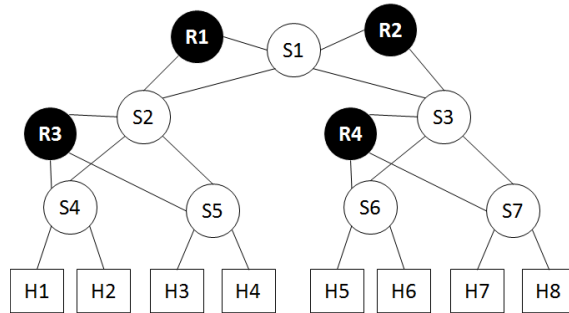


Figure 54: Experiment topology

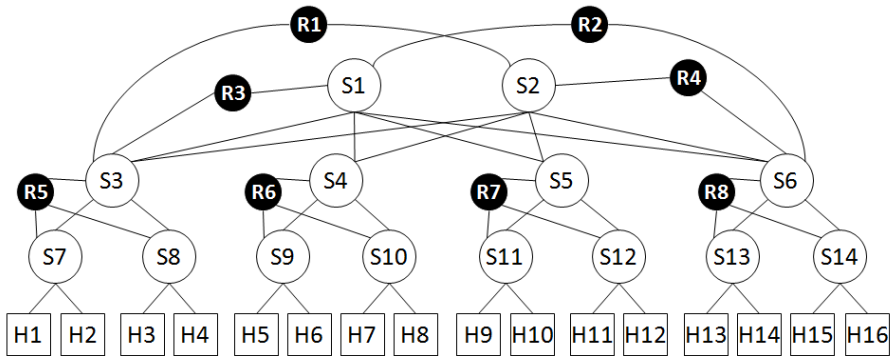
of testbed experiment and emulation along with topology and metrics. We then compare overall and per-topology performance and overhead of SoftDance with others. Finally, we contrast SoftDance with combined existing techniques.

We deployed a testbed experiment to verify that SoftDANCE works practically in a physical testbed system. As shown in Figure 54, the experiment consists of 3 clients ($H1, H2$, and $H3$), a server ($H4$), 4 SDMBs ($R1$ to $R4$), OpenVSwitches, and a controller. The controller is connected to all nodes through an out-of-band network (not shown in figure). There are three paths and each client sends the same dataset as other clients. Thus, redundancy of all datasets is $\frac{2}{3}$.

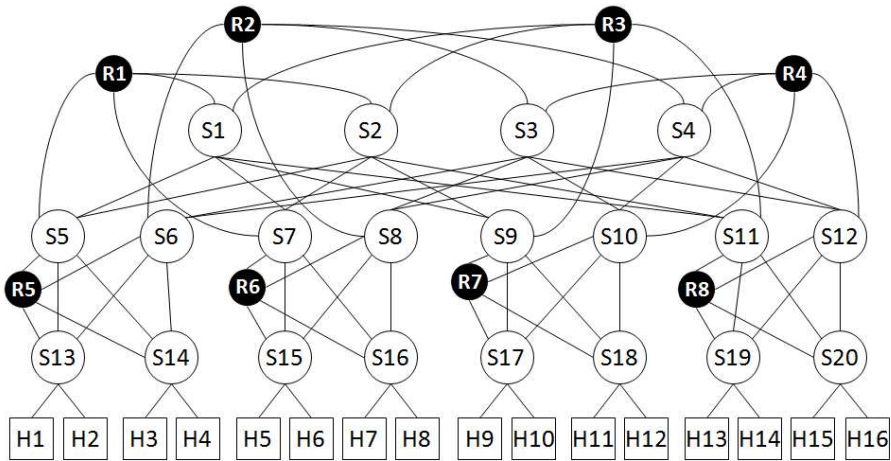
We also set up several topologies such as a tree, a multi-rooted tree, and a fat-tree based on mininet [51] as shown in Figure 55. The purpose of choosing the topologies is to validate SoftDANCE on typical topologies in Data Center Network (DCN). In all



(a) Tree



(b) Multi-rooted tree



(c) Fat-tree

Figure 55: Emulation topology

topologies, we choose a server that is on the far right side: *H8* in tree, *H16* in multi-tree and fat-tree. Other hosts act as clients. $R\{x\}$ is a SDMB and $S\{x\}$ is an OpenVSwitch. A controller communicates with all nodes through in-band (not shown in figure). We do not use multi-path because our main purpose is to measure performance and overhead on the same path for all compared techniques: a switch is chosen by a spanning tree in both multi-tree and fat-tree. Thus, number of nodes selected for multi-tree and fat-tree are same with a single core switch. Like experiment, all clients send the same dataset as other clients, so redundancy is $\frac{6}{7}$ for tree and $\frac{14}{15}$ for multi-rooted tree/fat-tree.

We use two metrics for measuring performance: storage space saving and network bandwidth saving, and other two metrics for measuring overhead: processing time and memory size. To present storage space saving, we use deduplication ratio. Deduplication ratio is a typical means to show how much storage space is reduced, and is computed by $\frac{\text{volume of redundant data eliminated}}{\text{total volume sent by all clients}} * 100$. Network bandwidth saving is computed by $\frac{\text{Reduced traffic size}}{\text{Total traffic size without redundancy elimination}} * 100$. For overhead metrics, we measure processing time occurred at clients, a server, and RE boxes. We also measure the size of memory that clients, a server, and RE boxes hold.

For the dataset, we use campus log data that has been captured at a university data center. The log data are backed up to storage servers every week. The used dataset has rare redundancies under 2%. Thus, for all techniques used, intra-redundancy ratio found when a single client sends a dataset is under 2% at maximum.

SoftDance is compared with client Dedup, server Dedup, and network wide RE

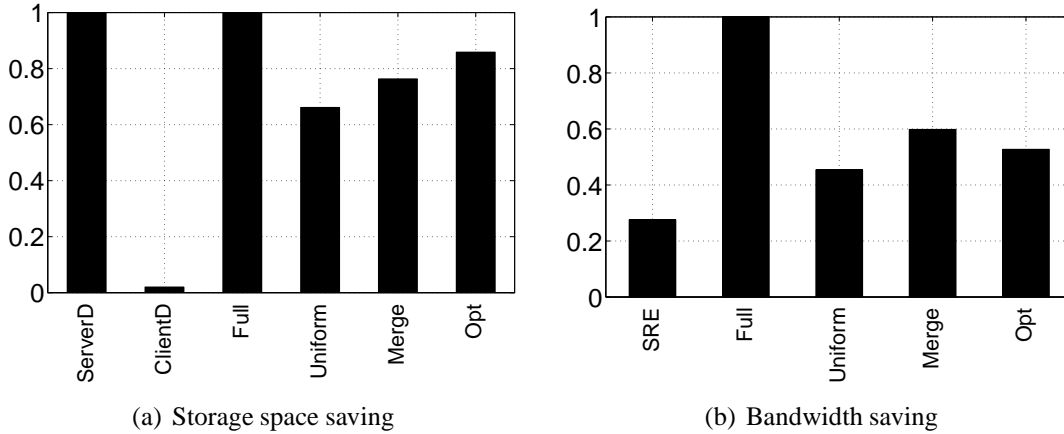


Figure 56: Comparison of performance

(SmartRE). We implemented existing techniques for comparison. Client Dedup is denoted as ClientD. Server Dedup is divided into file-granularity Dedup (File Dedup), fixed-size block granularity Dedup (Fix Dedup), and variable-size chunk granularity Dedup (Var Dedup) based on granularity. SmartRE distributes hash ranges based on its optimization LP [5]. SmartRE is denoted as SRE. We also compare our approaches including SD-full, SD-uniform, SD-merge, and SD-opt.

6.8 Storage Space and Network Bandwidth Saving

We present overall performance of SoftDance compared to existing techniques across all evaluation topologies: experiment and emulation (tree and multintree/fat-tree). For this purpose, we compare relative value normalized for all metrics as in Figures 56 and 57. We use log scale for Figures 57(a) and 57(b) because the gap between largest and smallest one is huge, and multiply 10^4 and 10^3 respectively to read figures easily.

For storage space saving as shown in Figure 56(a), SoftDance shows the closest

performance to server Dedup (ServerD) that is the best for saving storage space in existing techniques. SD-full shows exactly the same space saving as server Dedup. This indicates SD-full does not miss any redundancy through network. SD-uniform, SD-merge, and SD-opt have close space saving to the server Dedup. Meanwhile, client Dedup is the worst for space saving and has only 1.6% saving by eliminating redundancy inside a client. This shows client Dedup does not deal with redundancy across clients. SmartRE does not contribute storage space saving because it runs only on network in an application agonistic fashion. In SoftDance approaches, SD-merge shows better performance than SD-uniform due to finding more redundancies at merge SDMBs.

For bandwidth saving in Figure 56(b), SoftDance shows 2-4x more bandwidth saving than SmartRE. The reason that SmartRE shows lower bandwidth saving than SoftDance is that SmartRE fails to find inter-path redundancies passing different ingress routers that are encoders. For example, in Figure 54, two duplicate packets that arrive encoders ($R1$ and $R2$) from different hosts ($H1$ and $H2$) are determined to be unique, and thereafter traverse to a server without eliminating redundancy as if they were unique packets. To investigate our argument, we choose a multi-rooted tree as shown in Figure 55 and build a test case where edge SDMB is connected to only a client (e.g., $R5$ is connected to only $H1$ but not with $H2, H3$, and $H4$). We choose 4 clients ($H1, H5, H9, H13$) and one server ($H16$). The bandwidth saving on the test case shows only about 4% that is summation of intra-path redundancy from each client: 70% inter-path redundancy is not detected ($\frac{3}{4}(75\%) - 4\%$).

In SoftDance, SD-opt (optimize) shows close storage space saving to SD-full considering resource constraints, but has lower bandwidth saving than SD-merge. Note that SD-opt optimizes based on matches between packets, which seeks overall benefit of space and bandwidth saving rather than only a single benefit. Thus, a single benefit can be lower than heuristic approaches.

6.9 CPU and Memory Overhead

For processing time in Figure 57(a), SoftDance (denoted as Full) is the lowest among all techniques. Client Dedup (denoted as ClientD) and Var Dedup (denoted as S-Var) have 100x and 10x higher processing time than SmartRE due to expensive chunking. Even SmartRE (denoted as SRE) shows larger processing time than SoftDance because of the sliding fingerprinting. For memory size in Figure 57(b), SoftDance (denoted as Full) has 40x less memory than SmartRE. This is because SoftDance only stores indexes but SmartRE saves packets as well as indexes in caches for encoders and decoders. Evicting indexes and packets from caches can reduce memory size, but may lead to low bandwidth because same packets of evicted ones are not encoded (that is, they are found to be unique). However, SoftDance (Full) still consumes larger memory than server Dedup and client Dedup due to more indexes on fine-grained granularity (SoftDance uses 1.5 KB packet payload, but server Dedup and client Dedup use 8 KB chunk(or block) granularity). To reduce memory size, SoftDance distributes indexes based on hash-based sampling [68]. Figure 58 demonstrates that memory size (concretely index size) required by SD-full can be reduced up to 3x by SD-opt.

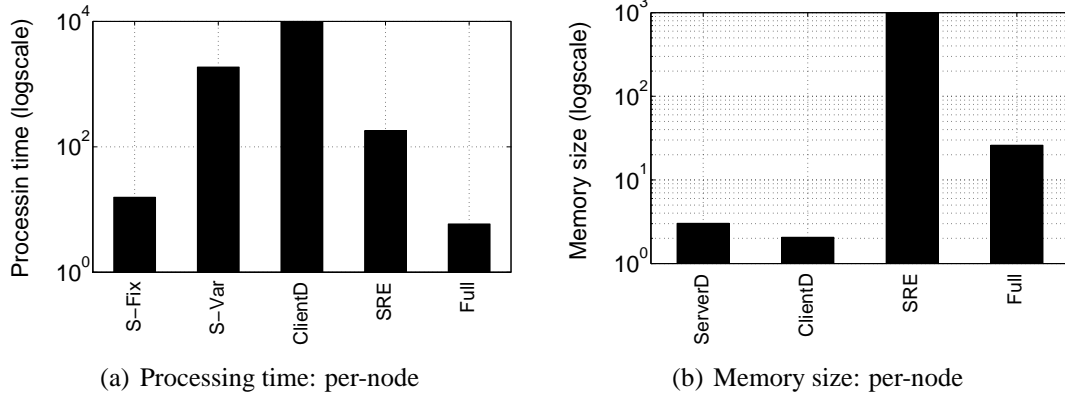


Figure 57: Comparison of overhead

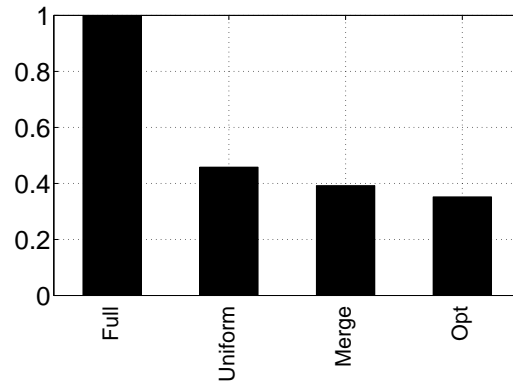


Figure 58: Memory size among SoftDance approaches: per-node

6.10 Performance and Overhead per Topology

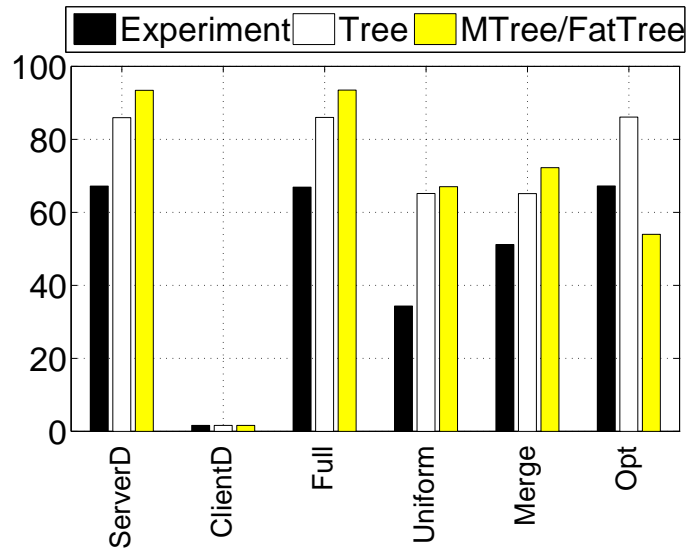
We show performance per metric and changes of performance depending on different topologies. The differences among topologies are two fold: number of clients and location of clients. First, the number of clients in tree topology is more than that in experiment, resulting in more redundancy because each client sends the same dataset as other clients in our evaluation. Second, in experiment, clients are attached to both edge and interior switches while in emulation, clients are attached to only edge switches. Our focus

is what performance each technique acquires on the differences.

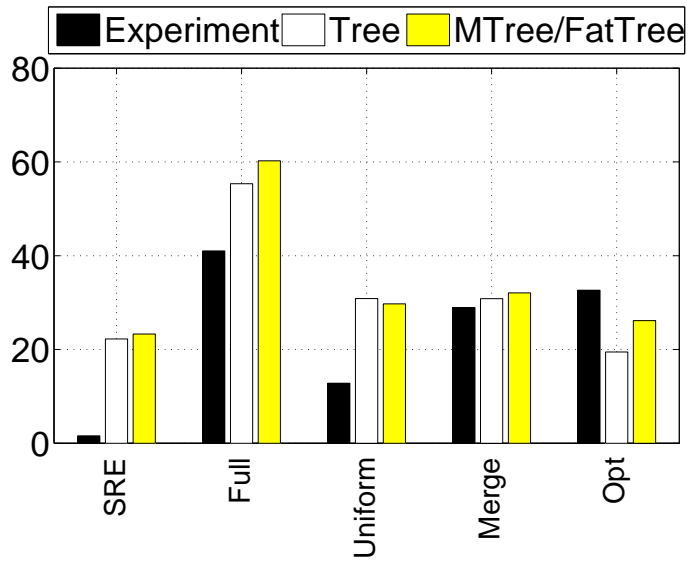
Storage space saving increases as the number of clients increases for all techniques (except for SD-opt) as shown in Figure 59(a). This is because redundancies increase proportionally to the number of clients that send same dataset. SD-full shows exactly the same space saving as server Dedup, which indicates SD-full does not miss any redundancies through the network. In experiment, SD-merge shows higher space saving than SD-uniform, which indicates a merge SDMB has more redundant packets (originated from different paths) than a forwarding node with one incoming degree. SD-opt achieves the most space saving among distributed indexing approaches. However, we find SD-opt in multi-tree/fat-tree shows a bit anomalous result which was expected to be higher than in tree topology. We are investigating the anomaly.

For bandwidth saving in Figure 59(b), SoftDance shows more bandwidth saving than SmartRE. In experiment, SmartRE has much less bandwidth saving by 10-40x times than SoftDance while in tree (or mtree/fattree), SmartRE has 2-3x less bandwidth saving than SoftDance. The significant difference between in experiment and in tree(or mtree/fattree) for SmartRE is not caused by the increase in the number of clients but by a fact that SmartRE fails to find inter-path redundancies passing different ingress routers.

As the number of clients increases, processing time increases as shown in Figure 60(a) for all techniques (except for SmartRE). However, the velocity of change is different; SoftDANCE (denoted as Full) increases more slowly in processing time than others. Other SoftDance approaches such as SD-uniform, SD-merge, SD-opt (not shown here) have the almost same processing time as SD-full. Client Dedup and variable-size

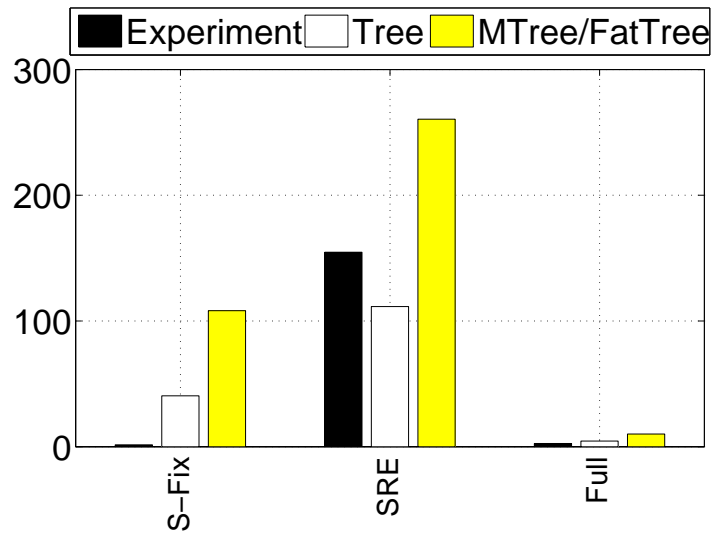


(a) Storage space saving (%)

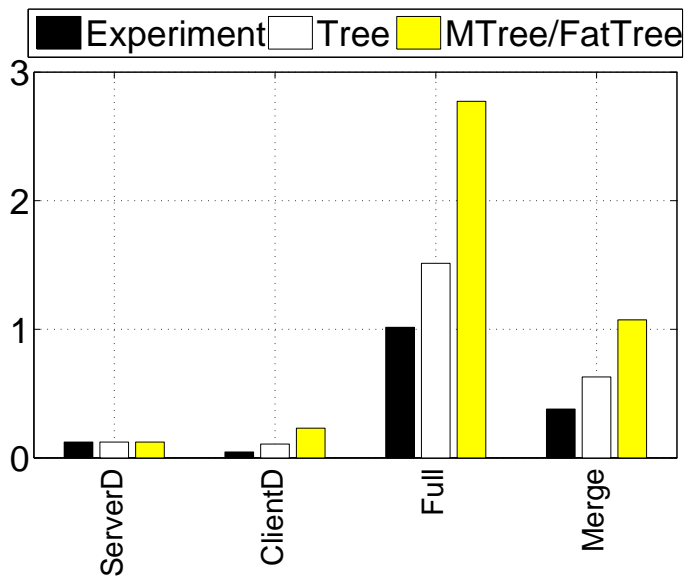


(b) Bandwidth saving (%)

Figure 59: Performance per topology



(a) Processing time: per-node (sec)



(b) Memory size: per-node (MB)

Figure 60: Overhead per topology

server Dedup are not shown as readable figures due to excessive processing time. Interestingly, in experiment, SmartRE shows larger processing time than in tree topology. We find that computers used for REboxes in experiment are much slower than a computer used for emulation (tree and multi-tree/fat-tree), which amplifies processing time slowed by fingerprinting. Memory size increases proportional to increase in the number of clients as shown in Figure 60(b). For SmartRE (not shown in figure), memory size in SDMBs is X , $1.1X$, $2.5X$ for experiment, tree, and multi-tree/fat-tree respectively where X requires 40 times more memory than SD-full. SD-full has more index size than server Dedup and client Dedup, but SoftDance can reduce the indexes by utilizing an indexing scheme like SD-merge to reduce memory size of SD-full in the figure.

6.11 SoftDance vs Combined Existing Deduplication Techniques

We tested some of scenarios when client data is transferred across network links to be stored in a server, while each Dedup and NRE can be performed for a benefit of its own domain. The data may go through various forms of deduplication processes redundantly that may incur significant processing and memory overhead. To compare with SoftDance, we envision two combined approaches that can be used as a network and storage service using existing techniques: client Dedup (storage service) + SmartRE (network service) and server Dedup (storage service) + SmartRE (network service).

For storage space saving as shown in Figure 61(a), SoftDance shows the best space saving equal to “ServerD+SRE”. Both two combined approaches rely on storage services including client Dedup and server Dedup because SmartRE is not applicable for storage

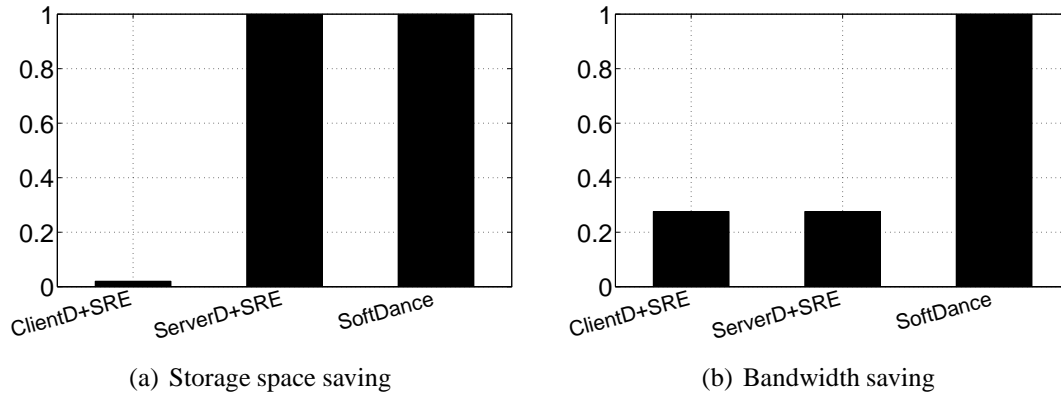


Figure 61: Performance of combined approaches

space saving. For bandwidth saving as shown in Figure 61(b), SoftDance saves the most bandwidth compared to two combined approaches. For two combined approaches, bandwidth saving is determined by performance of SmartRE. For processing time, SoftDance outperforms the two combined approaches as shown in Figure 62(a). The slow processing time of the two combined approaches is due to expensive chunking and fingerprinting. For memory size as shown in Figure 62(b), SoftDance requires less memory size than the two approaches combined. This is attributed to the fact that SmartRE stores packets itself as well as indexes. The slight reduction in memory size inside a client becomes invalid due to excessive memory size needed by SmartRE. Overall, the evaluation results show that in the scenarios of both end-systems and networks performing deduplication redundantly, SoftDance achieves very efficient processing and memory overhead.

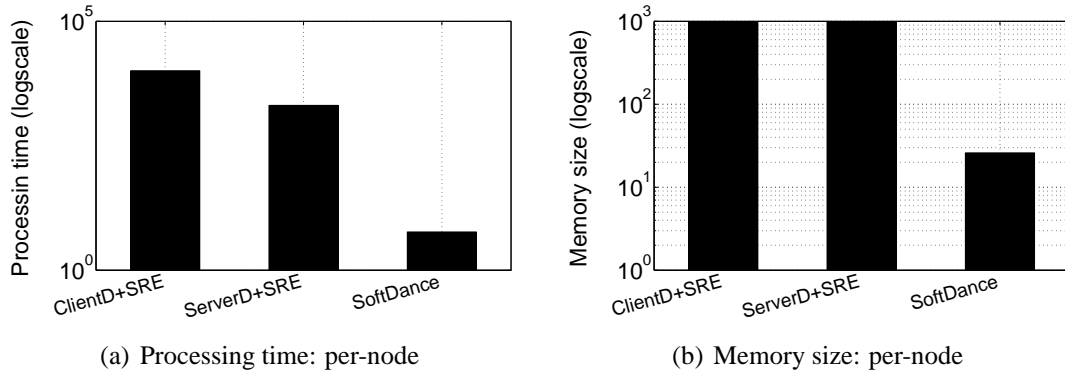


Figure 62: Overhead of combined approaches

6.12 Summary

In this chapter, we proposed SoftDance, an efficient software-defined deduplication as a network and a storage service that gives both storage space savings and network bandwidth savings while significantly reducing processing time and memory size. We developed efficient encoding and indexing algorithms for a SoftDance middlebox (SDMB) and an effective control mechanism for an SDN controller. We also built a prototype of testbed experiments and Mininet-based emulations to evaluate SoftDance on real system environments and typical DCN topologies. Our evaluation results show that SoftDance saves 2-4x more bandwidth than an RE technique (SmartRE) and same/close storage space saving to the Dedup technique with low overhead, while achieving very efficient processing and memory overhead.

CHAPTER 7

MOBILE DE-DUPLICATION

In this chapter, we show an enhancement of client-based deduplication for popular files in mobile devices, where each file is deduplicated based on the file's structure considering low capacity mobile device. We also consider security of the files, and observed performance of encryption algorithms based on systems with different CPU types such as android mobile device (ARM-CPU) and desktop Linux server (Intel-CPU).

7.1 Large Redundancies in Mobile Devices

Currently, massive files are popularly created and used in mobile devices. Large amount of documents and image files are generated and used in mobile devices. Also watching video streams is one of major usages in mobile devices.

We address two issues. The first issue is that large redundancies exist in files of mobile devices. For example, nowadays there is a very popular application to take pictures of moving objects, called burst shooting mode. In this mode, we can take 30 pictures within a second and choose good pictures or remove bad ones, but this application may experience large redundancies between similar pictures. Also, a video file consists of I-frame that has images and P-frame that has delta information between I-frames. In scenes where actors keep talking in the same background, large portions of background become redundant: that is, I-frame has large redundancies that can be removed. The second

issue is that security becomes critically important in deduplication in mobile devices, and encryption function should be fast and consider low energy consumption considering the low capacity of mobile devices.

Thus, our approaches are to use structure aware deduplication for files based on files' structured formats with strong and fast encryption. We choose many different types of files including documents, emails, and image files in mobile devices. For structure aware deduplication, we decompose a file into objects, and deduplicate objects based on structure library where structure formats are defined. For security, the encryption algorithm has different performances and strengths. Generally, stronger encryption is slower due to more computation. Thus, we develop an idea that for more security-sensitive objects, strong encryption should be used, but for less security-sensitive objects, weaker encryption can be used for fast performance.

For varying systems with different CPU types, we measured the performance of strong encryption algorithms like Advanced Encryption Standard (AES) [17] and weaker encryption algorithms like Blowfish [67], Data Encryption Standard (DES) and 3DES [55], and RC2 [66]. Based on the results, we found that the performance of encryption can be effected by encryption strength as well as CPU types (Intel or ARM).

7.2 Approaches and Observations

We propose structured-based deduplication using encryption functions based on different level of security. Thus, in our approaches, we mainly focus on two purposes: how to efficiently decompose and reconstruct files in mobile devices, and how fast and

strong encryptions can be used for security of decomposed objects. For the first purpose, we are using structure aware deduplication that has been used for SAFE. we are investigating how efficiently it deduplicates images and video files in mobile devices, and we will focus more on the second purpose for security. The security and privacy issues on deduplication are discussed in many studies [30] [31] [32] [53].

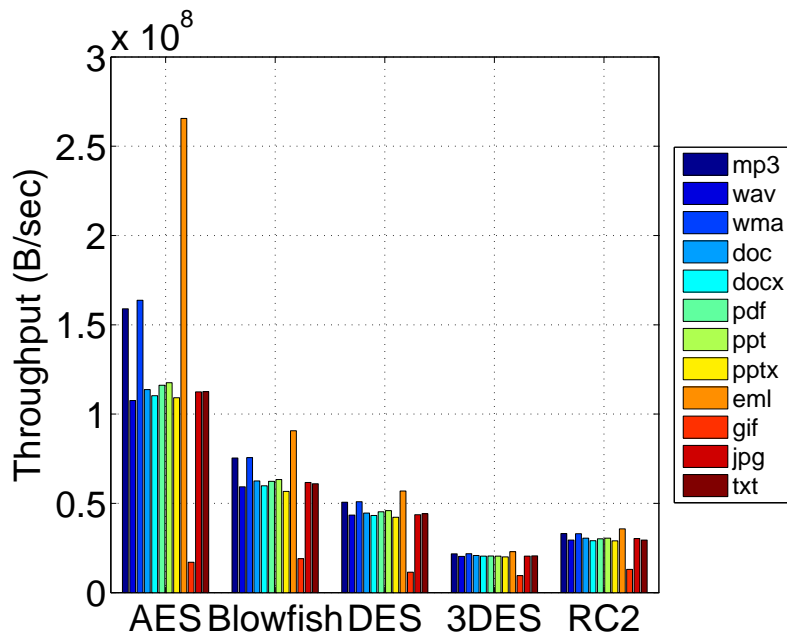
7.3 JPEG and MPEG4

JPEG [35] is popular compression technique for digital photography, and current mobile devices use JPEG as a default image format due to its small footprint compared to other image files. JPEG uses efficient compression algorithm such as Discrete Cosine Transformation (DCT). We argue that JPEG efficiently reduces redundancies of a single image, and our approach reduces redundancies among similar images. MPEG4 [36] is a popular compression for audio and video files. For example, streaming files in youtube are MPEG4.

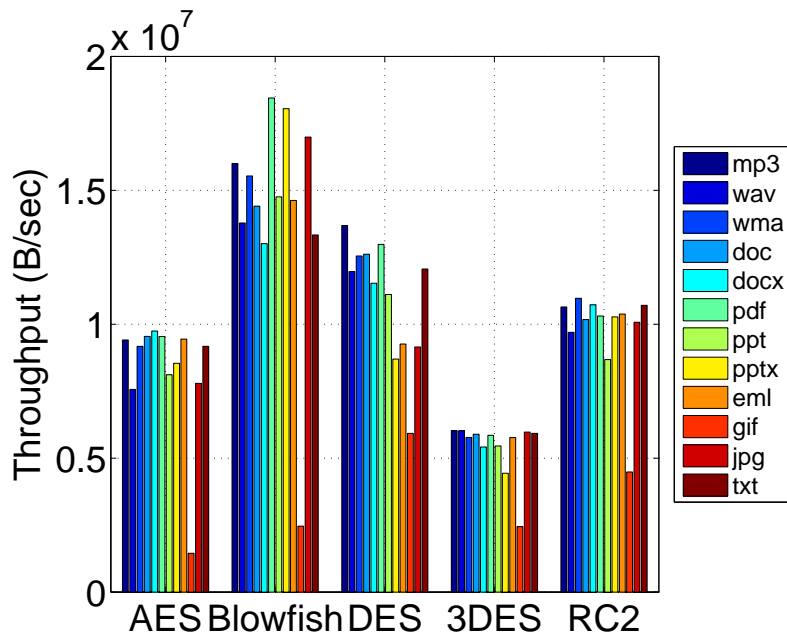
7.4 Throughput and Running Time of Encryption Algorithm

We show how encryption algorithms are performed depending on different file types and systems with different CPU types. Overall, AES outperforms other compared encryption algorithms on Intel-based system in terms of performance. However, for ARM-based systems like smart phone, blowfish shows the best performance among other algorithms.

As shown in Figures 63 and 64, we observed that encryption algorithms show

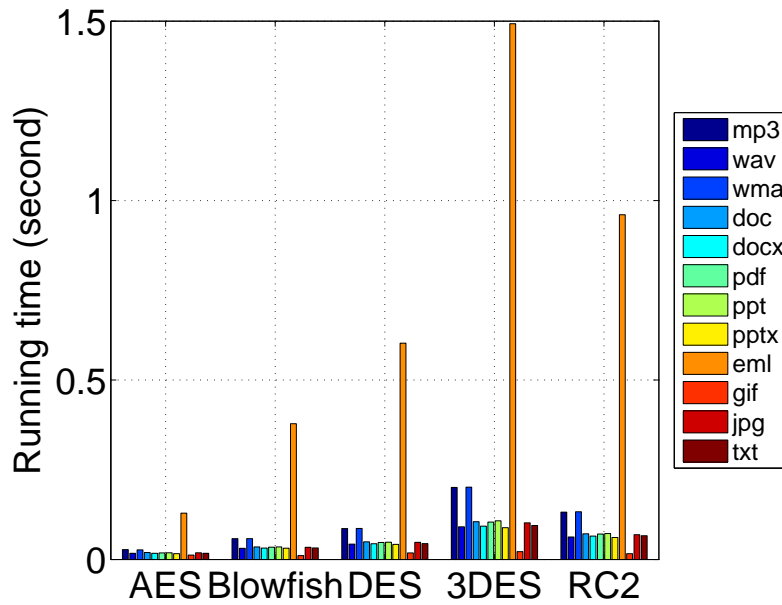


(a) Linux (Intel I5)

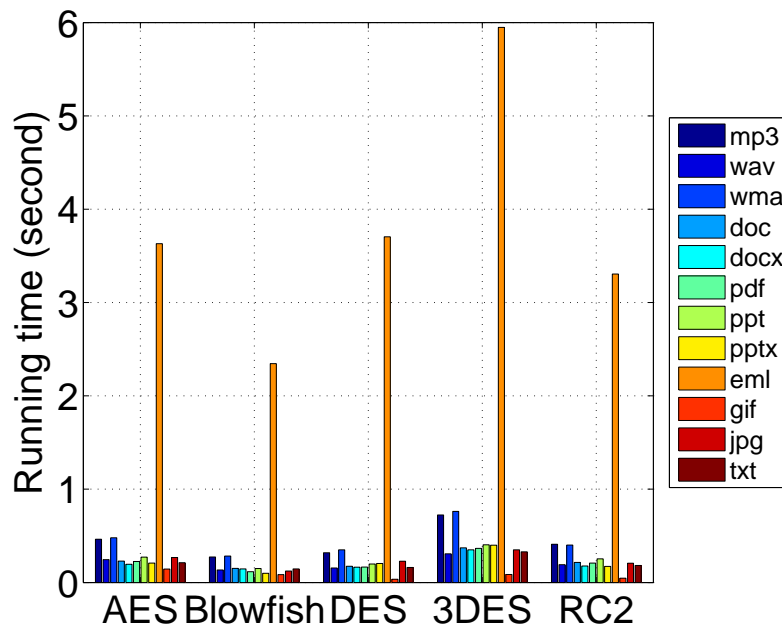


(b) Nexus 7 (ARM)

Figure 63: Throughput of encryption algorithms per file type



(a) Linux (Intel I5)

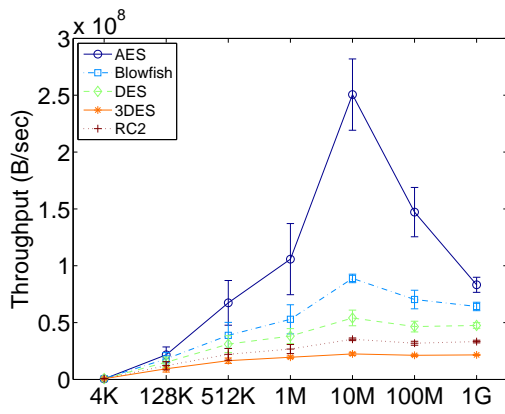


(b) Nexus 7 (ARM)

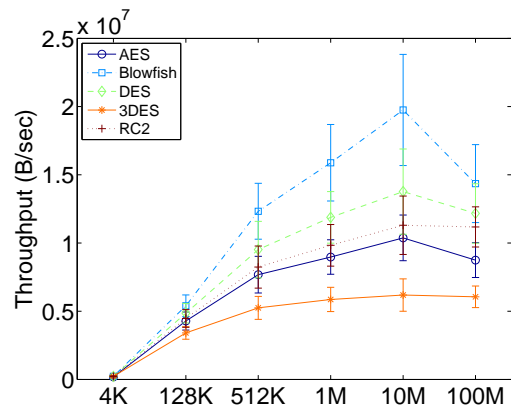
Figure 64: Running time of encryption algorithms per file type

different performances per system with different CPU types. As shown in Figure 63, we measured throughput of encryption algorithms in a Linux machine with Intel I5 and in a Nexus 7 with ARM, for different file types including audio, document, image, text, and email files. We found that in Linux, AES outperforms other encryption algorithms, but in nexus 7, blowfish has the highest performance. We were interested in this result because a previous study [19] insisted that blowfish always has the best performance. The reason of AES's best performance in Linux system is that Intel CPU has hardware-support instruction sets for AES; that is call AES New Instruction (AES_NI). In the same vein, Figure 64 presents that AES is the fastest in Linux system but blowfish is the fastest in Nexus7.

We measured throughput and processing time of encryption algorithms in Linux and nexus 7, varying the data size from 4 KB to 1 GB. Overall, In both systems, throughput and processing time increase as the growth of data size. However, for throughput as shown in Figure 65, there is a threshold size (here, 10 MB) after which throughput decreases. For processing time as shown in Figure 66, very small data (like 4 KB) takes longer time than relatively larger data (like 128 KB). However, the processing time from 4 KB to 128 KB decreases for both AES and blowfish in both systems. These results show the importance of choosing granularity of deduplication, considering performance of encryption in deduplication. 128KB is 32 4KBs. Thus, encrypting a 128 KB object is 32 times faster than encrypting 32 4KB objects. However, using 128KB granularity finds less redundancies using 4 KB granularity. As a result, we need to select a granularity considering balance between removing redundancies and encryption processing time.

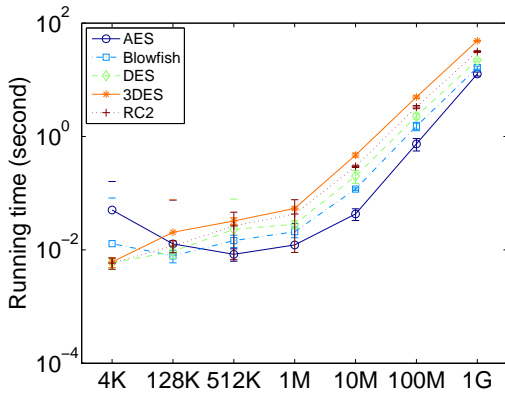


(a) Linux (Intel I5)

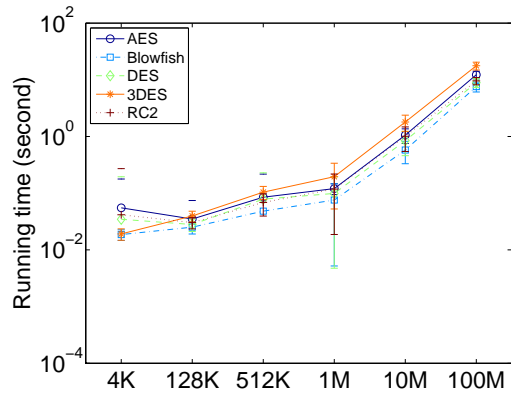


(b) Nexus 7 (ARM)

Figure 65: Throughput of encryption algorithms per file size



(a) Linux (Intel I5)



(b) Nexus 7 (ARM)

Figure 66: Running time of encryption algorithms per file size

7.5 Summary

In this chapter, we show a client-based deduplication for popular files in mobile devices, mobile deduplication that remove redundancies of the files in mobile devices. Considering low capacity of mobile devices, we propose to use structure aware deduplication for the files to improve processing time. Also, we observed that the performance of encryption is changed depending on the strength level of security as well as systems with different CPU types. In future work, we are investigating efficient structure aware deduplication for JPEG [35] and MPEG4 [36] files in terms of storage space savings and processing time overhead.

CHAPTER 8

CONCLUSIONS

In the era of data explosion, huge redundancies exist in storages and networks. Existing deduplication solutions such as storage data deduplication and network redundancy elimination are not as efficient as possible to optimize data moving from clients to servers through networks.

Thus, my contribution is devoted to develop an efficient deduplication framework to optimize data in a chain from clients to servers through network, and to make components for the framework. We developed the components such as Hybrid Email Deduplication System (HEDS) on the server side, Structure Aware File and Email Deduplication for Cloud-based Storage Systems (SAFE) on the client side, and Software-Defined Deduplication as a Network and Storage Service (SoftDance) on the network side for the deduplication framework. HEDS efficiently achieves an trade-off of file-level and block deduplication for email systems. SAFE a exploits structure-based granularity rather than using physical chunk granularity, which enables SAFE to as fast as file-level deduplication and has the same space savings as block deduplication with a low overhead. SoftDance, as an in-network deduplication, chains storage data deduplication and network redundancy elimination functions by using Software Defined Network (SDN), and it achieves storage space savings and network bandwidth savings with low processing time and memory overhead in storages and networks. We are also working on mobile deduplication with

popular files such as image and video files in mobile devices.

For the future work, we are planning to deploy and explore more reliability for network dynamics, storage workload, and failure, as well as scalability in Cloud environments.

Bibliography

- [1] Adobe. ISO32000:Document management:Portable document format. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf.
- [2] Alvarez, C. NetApp deduplication for FAS and V-Series deployment and implementation guide (TR-3505). <http://www.netapp.com/us/media/tr-3505.pdf>.
- [3] Amazon. Amazon Simple Storage Service. <http://aws.amazon.com/s3/>.
- [4] Anand, A., Gupta, A., Akella, A., Seshan, S., and Shenker, S. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. of the ACM SIGCOMM 2009 conference on Data communication* (2008).
- [5] Anand, A., Sekar, V., and Akella, A. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proc. of the ACM SIGCOMM 2009 conference on Data communication* (2009).
- [6] Bloom, B. H. Space/time Trade-offs in Hash Coding with Allowable Errors. vol. 13, *Communication of the ACM*.
- [7] Bolosky, W., Corbin, S., Goebel, D., and Douceur, J. Single instance storage in Windows 2000. In *Proc. of the 4th USENIX Windows Systems Symposium* (2000).
- [8] Bonwick, J. ZFS deduplication. https://blogs.oracle.com/bonwick/entry/zfs_dedup.

- [9] Cisco. Wide Area Application Services. <http://www.cisco.com/c/en/us/products/routers/wide-area-application-services/index.html>.
- [10] Citrix. CloudBridge. <http://www.citrix.com/products/cloudbridge/overview.html>.
- [11] Debnath, B., Sengupta, S., and Li, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *USENIX Annual Technical Conference* (2010).
- [12] Dong, W., Douglass, F., Li, K., Patterson, R. H., Reddy, S., and Shilane, P. Trade-offs in Scalable Data Routing for Deduplication Clusters. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2011).
- [13] Drago, I., Mellia, M., M Munafo, M., Sperotto, A., Sadre, R., and Pras, A. Inside dropbox: understanding personal cloud storage services. In *Proc. of the 2012 ACM conference on Internet measurement conference (IMC)* (2012), pp. 481–494.
- [14] Dropbox. <http://www.dropbox.com>.
- [15] Dropbox. REST API. <https://www.dropbox.com/developers/core/docs>.
- [16] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., and Welnicki, M. HYDRAsTOR: A Scalable Secondary Storage. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2009).

- [17] Dworkin, M. J., Barker, E. B., Nechvatal, J. R., Foti, J., Bassham, L. E., Roback, E., and Jr, J. F. D. Advanced Encryption Standard. http://www.nist.gov/manuscript-publication-search.cfm?pub_id=901427.
- [18] ECMA. Standard ECMA-376 : Office Open XML File Formats. <http://www.ecma-international.org/publications/standards/Ecma-376.htm>.
- [19] Elminaam, D. S. A., Kader, H. M. A., and Hadhoud, M. M. Performance Evaluation of Symmetric Encryption Algorithms. *International Journal of Computer Science and Network Security (IJCSNS)* 8, 12 (2008), 280–286.
- [20] ElShimi, A., Kalach, R., Kumar, A., Oltean, A., Li, J., and Sengupta, S. Primary Data Deduplication-Large Scale Study and System Design. In *USENIX Annual Technical Conference* (2012).
- [21] EMC. Achieving storage efficiency through EMC Celerra data deduplication. <http://china.emc.com/collateral/hardware/white-papers/h6265-achieving-storage-efficiency-celerra-wp.pdf>.
- [22] EMC. Avamar. <http://www.emc.com/backup-and-recovery/avamar/avamar.htm>.
- [23] EMC. Centera: Content Addresses Storage System, Data Sheet. <http://www.emc.com/collateral/hardware/data-sheet/c931-emc-centera-cas-ds.pdf>.
- [24] EMC. Networker. <http://www.emc.com/domains/legato/index.htm>.
- [25] Floodlight. Floodlight Module. <https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Module+Applications>.

- [26] Floodlight. Floodlight SDN Controller. <http://www.projectfloodlight.org/floodlight/>.
- [27] Freed, N., and Borenstein, N. S. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. <http://tools.ietf.org/html/rfc2045>.
- [28] FUSE. File in UserSpace. <http://fuse.sourceforge.net/>.
- [29] Guo, F., and Efstathopoulos, P. Building a High-performance Deduplication System. In *USENIX Annual Technical Conference* (2011).
- [30] Halevi, S., Harnik, D., Pinkas, B., and Shulman-Peleg, A. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), CCS '11, ACM, pp. 491–500.
- [31] Harnik, D., Pinkas, B., and Shulman-Peleg, A. Side Channels in Cloud Services: Deduplication in Cloud Storage. *Security Privacy, IEEE* 8, 6 (2010), 40–47.
- [32] Hu, W., Yang, T., and Matthews, J. N. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review* 44, 3 (Aug 2010), 110–115.
- [33] IBM. IBM White paper: IBM StorageTank - A distributed storage system. <https://www.usenix.org/legacy/events/fast02/wips/pease.pdf>.
- [34] IDC. The Digital Universe in 2020. <http://idcdocserv.com/1414>.
- [35] ISO. JPEG, digital compression and coding of continuous-tone still images. http://www.iso.org/iso/catalogue_detail.htm?csnumber=18902.

- [36] ISO. MPEG4, Coding of audio-visual objects: Part 12: ISO base media file format.
http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=38539.
- [37] ISO, and IEC. ISO/IEC 29500-1:2008. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51463.
- [38] Json.org. C++ JSON parser. <http://sourceforge.net/projects/jsoncpp/>.
- [39] JustCloud. <http://www.justcloud.com/>.
- [40] Kim, D., and Choi, B.-Y. HEDS: Hybrid Deduplication Approach for Email Servers. In *Ubiquitous and Future Networks (ICUFN), 2012 Fourth International Conference on* (2012).
- [41] Kim, D., Song, S., and Choi, B.-Y. SAFE: Structure-Aware File and Email Deduplication for Cloud-based Storage Systems. In *Proc. of the 2nd IEEE International Conference on Cloud Networking* (Nov 2013).
- [42] Klimt, B., and Yang, Y. The enron corpus: A new dataset for email classification research. 217–226.
- [43] Li, J., He, L.-w., Sengupta, S., and Aiyer, A. *Multimodal Object De-duplication*. Microsoft Corporation, 08 2009. Patent.
- [44] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2009).

- [45] Liu, C., Lu, Y., Shi, C., Lu, G., Du, D., and Wang, D. ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System. In *Storage Network Architecture and Parallel I/Os, (SNAPI) Fifth IEEE International Workshop on (2008)*, pp. 29–35.
- [46] Meyer, D. T., and Bolosky, W. J. A Study of Practical Deduplication. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST) (Feb 2011)*.
- [47] Microsoft. Exchange server 2003. <http://technet.microsoft.com/en-us/library/bb123872%28EXCHG.65%29.aspx>.
- [48] Microsoft. Exchange server 2007. <http://www.microsoft.com/exchange/en-us/exchange-2007-overview.aspx>.
- [49] Milter.org. Sendmail Mail Filters. <https://www.milter.org/home>.
- [50] Min, J., Yoon, D., and Won, Y. Efficient Deduplication Techniques for Modern Backup Operation. In *IEEE Transactions on Computers (June 2011)*, vol. 60, pp. 824–840.
- [51] Mininet.org. Mininet. <http://mininet.org/>.
- [52] Mozy. <http://mozy.com/>.
- [53] Mulazzani, M., Schrittwieser, S., Leithner, M., Huber, M., and Weippl, E. Dark clouds on the horizon: using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX conference on Security (2011)*, SEC’11.

- [54] Muthitacharoen, A., Chen, B., and Mazières, D. A low-bandwidth network file system. In *SOSP* (Dec. 2001).
- [55] National Institute of Standards and Technology (NIST). Data Encryption Standard (DES). <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>.
- [56] National Institute of Standards and Technology (NIST). Secure Hash Standard 1 (SHA1). <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.
- [57] National Institute of Standards and Technology (NIST). Secure Hash Standard 256 (SHA256). <http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf>.
- [58] NEC. Hydrastor. <https://www.necam.com/hydrastor/>.
- [59] Netfilter.org. iptables extensions. <http://ipset.netfilter.org/iptables-extensions.man.html>.
- [60] Netfilter.org. libnetfilter_queue. http://www.netfilter.org/projects/libnetfilter_queue/.
- [61] Pagh, R., and Rodler, F. F. Cuckoo Hashing. *J. Algorithms* 51, 2 (May 2004), 122–144.
- [62] PKWARE. ZIP File Format Specification. <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>.
- [63] Quinlan, S., and Dorward, S. Venti: A New Approach to Archival Storage. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (Jan. 2002).

- [64] Rabin, M. O. Fingerprinting by random polynomials. Tech. Rep. Report TR-15-81, Harvard University, 1981.
- [65] Riverbed. SteelHead for WAN Optimization. <http://www.riverbed.com/products/wan-optimization/>.
- [66] Rivest, R. A Description of the RC2(r) Encryption Algorithm, RFC2268. <https://www.ietf.org/rfc/rfc2268.txt>.
- [67] Schneier, B. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop* (London, UK, UK, 1994), Springer-Verlag, pp. 191–204.
- [68] Sekar, V., Reiter, M. K., Willinger, W., Zhang, H., Kompella, R. R., and Andersen, D. G. CSAMP: a system for network-wide flow monitoring. In *NSDI* (2008).
- [69] Sendmail.com. Sendmail. http://www.sendmail.com/sm/open_source/.
- [70] Silverberg, S. SDFS. <http://opendedup.org>.
- [71] Spring, N. T., and Wetherall, D. A protocol-independent technique for eliminating redundant network traffic. In *Proc. of the ACM SIGCOMM 2000 conference on Data communication* (2000).
- [72] Srinivasan, K., Bisson, T., Goodson, G., and Voruganti, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. of the Tenth USENIX Conference on File and Storage Technologies (FAST)* (2012).

- [73] Stenberg, D. cURL. <http://curl.haxx.se/>.
- [74] Symantec. NetBackup. <http://www.symantec.com/netbackup>.
- [75] Symantec. PureDisk. <http://www.symantec.com/netbackup-puredisk>.
- [76] Xia, W., Jiang, H., Feng, D., and Hua, Y. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *USENIX Annual Technical Conference* (2011).
- [77] Yan, F., and Tan, Y. A Method of Object-based De-duplication. *Journal of Networks* 6, 12 (2011), 1705–1712.
- [78] Zhu, B., Li, K., and Patterson, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2008).

VITA

Daehee Kim graduated from Pusan National University in South Korea in 1995 as a bachelor of science for Computer Science. After graduation, he had worked for Information Technology (IT) companies including IBM Korea for more than 10 years as a programmer, a software engineer, a system architect, and a project leader. He performed various large scale system integration (SI) projects where he developed Web GroupWare system, Web Portal system, and Web Intranet applications using Java programming language. After he went abroad to study, he gained a Master of Science degree for Computer Science from State University of New York at Binghamton in 2008. At that time, he joined a project to develop a multicasting protocol for a wireless sensor network.

After one year's Ph.D. study in Wichita State University, he joined interdisciplinary Ph.D. curriculum in University of Missouri-Kansas City in 2009. His coordinating discipline is Telecommunication & Computer Networking and his co-discipline is Computer Science. His main research interest is to optimize data in storages and networks, and main technique is deduplication to reduce redundancies. His dissertation topic is tightly coupled with deduplication to optimize data (reduce data volumes by removing redundancies). Daehee also performed several multicasting projects for wireless sensor networks.

He gained Outstanding Ph.D. student award from Telecommunication & Computer Networking discipline in 2012, and is a member of Institute of Electrical and Electronic Engineers (IEEE).