

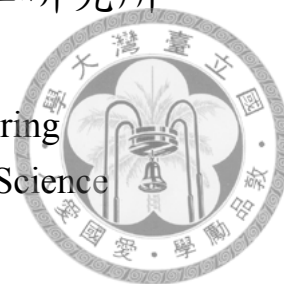
國立臺灣大學電機資訊學院電信工程學研究所

碩士論文

Graduate Institute of Communication Engineering
College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



FileFarm: 安全的雲中雲儲存系統

FileFarm: A Secured Cloud-of-Clouds Storage System

黃宇平

Yu-Ping Huang

指導教授：林宗男博士

Advisor: Tsung-Nan Lin, Ph.D.

中華民國 108 年 7 月

July, 2019

國立臺灣大學（碩）博士學位論文
口試委員會審定書



Filefarm: 安全的雲中雲儲存系統
Filefarm: A Secured Cloud-of-Clouds Storage System

本論文係黃宇平君（r06942065）在國立臺灣大學電信工程學研究所完成之碩（博）士學位論文，於民國 108 年 07 月 24 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

林 泉 男

（簽名）

（指導教授）

許怡中

蔡子淳

陳俊良

所 長

蘇火隆

（簽名）



誌謝

研究所第二年，撰寫論文的這段時光對我來說是求學生涯中特別踏實而難忘的一個篇章，這一年所經歷的種種求知、成長與蛻變讓向來倒頭就睡、不曾失眠的我，在口試的前一個晚上卻輾轉難眠了一整夜。當實驗、論文、簡報都已準備就緒並充分演練過，主宰我心的已非緊張不安的情緒，而是一種溫馨、喜悅與興奮摻雜而成的複雜情感：忙活了數個月，盡了一切該盡的努力後，我終於有時間回過頭來，想想那些生命中重要的人們，如何幫助我走過這段艱辛的研究旅程。

首先我要感謝人生中的導師 - 林宗男教授。在研究所兩年的時光中，老師幾乎每週都會找時間與我單獨討論研究進度，一路引導我定義問題、分析、提出解決方法、比較、設計實驗並觀察數據。我的研究主題並不是自己想出來或由老師指定的，而是在一次次的面談中逐漸由多個領域收斂出來的結果。謝謝老師因材施教，讓擅長實作多於理論的我能投入於這個應用層面較豐富的研究主題中；謝謝老師用心指導，每當我將修正完的投影片或新的實驗結果寄給老師，不管多忙碌，老師總會在隔日給我回覆或是找我前去討論；謝謝老師提供給我許多環境和資源，除了最好的經濟資助、最舒適的研究環境、最高級的設備外，我在研究之餘還有機會擔任電機系網多實驗、計算機程式、網路攻防實習等課程的助教，並且有許多機會親自準備教材、上台與同學分享我的所學；也謝謝老師持續提拔我走向下個階段。對於我和老師來說，我的碩士論文不只是一項研究，還是一個具有商業價值的技術，謝謝老師鼓勵我朝創業的方向前進，並且陪我持續思索可行的應用與解決方案。

接著我要感謝實驗室的夥伴：文于、煒傑、育維，我們從大二、大三就開始跟隨宗男教授，一起修課、做專題、當助教，一起分享許多生活瑣事以及研究上、學習上、工作上的心路歷程。儘管我們的研究主題相去甚遠，生活圈卻很接近。我們待在同一個實驗室中，各自忙著很不一樣的事情，卻都珍惜並讚嘆著彼此的能力。在畢業前兩三個月，我們一起為了論文與口試而忙碌，一起討論時程、論文格式以及各種需要注意的楣楣角角。在孤獨的研究之路上，能有這群相知相惜的朋友一路陪伴，真的是很爽快的事情！

除了老師和同學之外，我還要對同屬一個研究室的助理：小香姐、軒妤、柏盛、沛翰、佳宇、子建、合量，致上最深的感謝之意。每天早上我踏進研究室的那一刻開始，就充分地受到您們照顧，不論是安排與老師 meeting、借討論室、彙整研究進度報告、預定餐點、報帳、助教時數表簽章、添購設備、口試安排、專利申請、門禁設定、設備維修、VM 租借...各種疑難雜症都在您們的協助下才得以順利解決。這兩年來我們每天在實驗室碰面，各自為不同任務而忙碌著，時而抬起頭聊聊天、時而相互照應、偶爾一起出遊，兩年的時間在您們的陪伴下很快就過了，但相信我們會成為一輩子的朋友，謝謝你們！

最後我要感謝我的家人，在口試的當天，我正要出門，爸爸開著貨車從對向經過，輕輕按聲喇叭，搖下車窗靦腆地對我說聲：“加油！”這句簡短而熟悉的問候正是自有印象以來每當段考、基測、學測，爸爸都一定會對我說的一句話，十多年來始終如一！在繳交論文前兩三個月裡的每個深夜，我為了實驗數據而焦頭爛額時，媽媽總會在一旁追劇，時而偷瞥正在和螢幕上那一長串莫名程式碼奮鬥的我，陪我熬到半夜一兩點。在口試前一天，姐姐一如往常地命令我幫她買午餐、載她去搭火車、弟弟盧我幫他處理宿舍網路和民宿訂單的問題，他們一定是對我好，希望我平常心以對，嗯嗯一定是這樣的。

我要感謝所有在這段時間關心過我的家人、師長、朋友，給予我支持與寄託，讓我有機會全心全意地投入研究中，成為一個能夠發現、分析並解決問題的人。



摘要

在這篇論文中，我們描述 FileFarm: 一個建構於現有雲端儲存服務之上，為防止機密資料外洩、提升可靠性並去除對單一雲端依賴而設計的雲中雲儲存系統。為了解決既有雲中雲設計因集中式資料庫而造成的一致性和負載平衡問題，FileFarm 採取端對端 (P2P) 的解決方案。在 FileFarm 中，每個雲端服務皆為可獨立運作的單元，對客戶端提供相同的服務。這些被稱為 *Farmer* 的單元相互合作，共同組成一個端對端儲存網路。FileFarm 可容忍同時發生於至多 $K - 1$ 個 Farmer 上的錯誤，其中 K 為一個可調整的系統參數。當任何 Farmer 發生問題而無法提供服務時，FileFarm 系統會自動開啟一個修補機制，將資料備份到剩餘存活的 Farmer 上，確保每個資料區塊都在網路中被儲存了至少 K 份。為了在端對端網路中有效率地尋找資源，FileFarm 實作了 *Kademlia* 分散式雜湊表協定 [25]。FileFarm 從 *Kademlia* 中繼承了許多重要的特性，包含：(1) 備份數維護、(2) 高效率搜尋、(3) 負載平衡的設計。除此之外，作為一個企業級儲存系統，FileFarm 還需滿足以下四項條件：(1) 資料機密性 (2) 權限管理 (3) 成本效益 (4) 可存取性。為此，FileFarm 以此四個條件為面向分別設計對應的機制：(1) 加密與資訊分散演算法 (2) 分散式認證 (3) 儲存空間釋放與下載次序差異化 (4) 公有雲 ID 指定規則。我們基於系統所提供的特性將 FileFarm 與相關文獻進行比較，同時我們實作了一個系統原型並利用此原型進行一系列實驗以驗證我們聲稱的特性。此系統原型同時也是我們所提出的結構化端對端資料儲存解決方案之產品原型。

關鍵字：雲端儲存、雲中雲、分散式雜湊表、*Kademlia*、端對端儲存





Abstract

In this thesis, we describe FileFarm: a secured storage overlay that leverages existing cloud services to form a cloud-of-clouds storage system with better robustness, no single-point-of-failure and minimal data leakage concerns. To resolve the consistency and load-balancing issues caused by a centralized database design in conventional cloud-of-clouds work, FileFarm adopts a P2P strategy, in which each cloud operates as an independent node providing identical service for clients. The storage nodes, called *farmers*, cooperate with each other to form a peer-to-peer network, which tolerates concurrent failures occurring at any $K - 1$ farmers, where K is a configurable system-wise parameter. In case of failure occurring at any farmer, a storage repair procedure will be triggered automatically, which backs up data to surviving farmers and maintains at least K copies of each piece of data. To lookup resources efficiently in a P2P network, FileFarm implements *Kademlia* DHT(Distributed Hash Table) protocol[25]. Several desired properties of FileFarm are inherited from *Kademlia*: (1) redundancy maintenance, (2) efficient search and (3) load-balancing design. However, in order to serve as an enterprise-level storage, 4 further properties are required: (1) data confidentiality, (2) access management, (3) cost-efficiency, (4) retrievability. FileFarm meets these requirements by designing corresponding mechanisms, which collectively make FileFarm a robust, secure and cost-efficient storage solution: (1) *encryption* and *Information Dispersal Algorithm*, (2) *decentralized authentication*, (3) *storage release* and *prioritized*

download, (4)public farmer ID assignment. We compare FileFarm with related implementations in various aspects of properties. We also implement a proof-of-concept and perform a series of experiments on it to verify our claims. The proof-of-concept not only confirms our claims but also serves as a product prototype of our structured P2P file storage solution.



Keywords: Cloud Storage, Cloud-of-Clouds, MultiCloud, DHT, Kademia, P2P Storage



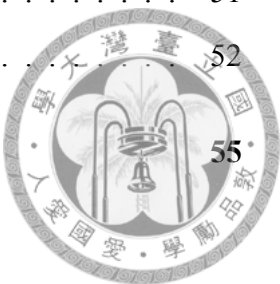
Contents

誌謝	iii
摘要	v
Abstract	vii
1 Introduction	1
1.1 Rise and Prominence of Cloud Storage	1
1.2 Concerns of Single Cloud Solution	1
1.3 FileFarm Overview	2
2 Related Work	5
2.1 Redundancy and Confidentiality	6
2.2 Location Query	7
2.3 Storage Repair	8
3 Background	11
3.1 Cloud Computing	11
3.2 Cloud Storage Service	12
3.3 Cloud-of-Clouds	13
3.4 Hybrid Cloud	13
3.5 Peer-to-Peer Systems	14
3.6 Distributed Hash Table	15
3.7 Kademlia	15

3.7.1	Load Balancing	15
3.7.2	Efficient Search	16
3.7.3	Redundancy Maintenance	20
3.8	Information Dispersal Algorithm	21
3.9	Public Key Infrastructure	22
4	Methodology	25
4.1	System Architecture	25
4.2	Application Models and Process Flows	26
4.3	DHT-Based Approach	29
4.4	Beyond Kademia	29
4.5	Data Confidentiality	30
4.6	Access Management	32
4.6.1	Decentralized Authentication	32
4.7	Cost Efficiency	35
4.7.1	Storage Release	35
4.7.2	Prioritized Download	36
4.8	Retrievability	38
4.8.1	Public Farmer ID Assignment	38
5	Experiments and Results	41
5.1	Environment	41
5.2	Experiment: NODE_LOOKUP Efficiency	42
5.3	Experiment: VALUE_LOOKUP Efficiency	43
5.4	Experiment: Retrievability	44
5.5	Experiment: Throughput	46
5.6	Experiment: Cost – Storage Release	48
5.7	Experiment: Cost – Prioritized Download	49
6	Conclusion	51
6.1	Problem	51



6.2 Approach	51
6.3 Evaluation	52
Bibliography	55







List of Figures

3.1	An example of Kademlia's Trie with ID length = 4	17
3.2	An example of IDA with schema $(p, q) = (4, 2)$	22
3.3	Procedure of certificate issuance and certificate-based authentication . . .	23
4.1	System architecture	25
4.2	Application model of client and farmer	27
4.3	Upload flow	28
4.4	Download flow	28
4.5	FileFarm's property stack	30
4.6	An example of encryption and sharding flow	31
4.7	An illustration of register procedure	34
4.8	An illustration of login procedure	34
4.9	Bit-reversal permutation ordering with prefix length = 4	39
5.1	NODE_LOOKUP steps with respect to K and network size n	42
5.2	VALUE_LOOKUP steps with respect to K and network size n	44
5.3	Retrievability of files with respect to α , K and q	45
5.4	Upload throughput with respect to file size and number of shards	47
5.5	Download throughput with respect to file size and number of shards . . .	47
5.6	Growth curve of static fee in 3 different settings: (1) migration with <i>storage release</i> , (2) migration without <i>storage release</i> , (3) no migration . . .	49
5.7	Comparison of transfer fee in 2 cases: (1) with <i>prioritized download</i> , (2) without <i>prioritized download</i>	50





List of Tables

1.1	Major pricing schema of 3 popular cloud storage providers	2
2.1	Comparison of properties provided by related cloud-of-clouds designs . .	5
5.1	Retrievability of files with respect to farmer's online probability α , network redundancy parameter K and IDA redundancy parameter q	45





Chapter 1

Introduction

1.1 Rise and Prominence of Cloud Storage

Over the past few years, cloud storage has experienced a metamorphosis from a trendy term to a mature technology and successful business. Due to its high reliability, low cost and minimum management overhead, more and more individual users and enterprises have chosen cloud storage as their major storage solution. According to a recent survey from Enterprise Storage Forum[32], 68% of the surveyed enterprises includes cloud in their current storage infrastructure and 35% of them store their company's primary storage data on cloud storage services. Clearly, high cost of storage hardware has lead businesses to outsource the headaches of in-house storage, and cloud storage turns out to be the best alternative.

1.2 Concerns of Single Cloud Solution

While cloud has rapidly become the dominant adoption among all kinds of storage choices, another aspect of concerns arises. According to a recent market survey done by Data-Core[1], security (57%), sensitive data (56%) and regulatory requirements (41%) are the 3 major concerns that blocks enterprises from migrating to a public cloud. With data outsourced to a third-party cloud provider, privacy settings are beyond the control of the enterprise. In this situation, sensitive data such as banking information or patient's medi-

cal records are exposed to the risks of being viewed or mishandled by malicious insiders in the single cloud. In addition, the "single cloud" solution also suffers from some potential risks such as service availability failure or vendor lock-in problem. For example, table 1.1 shows the major pricing schema of 3 popular cloud storage providers. Besides the monthly per-GB storage fee, consumers also need to pay a non-negligible fee for their per-GB download traffic. As a consequence, with a large amount of data stored on a single cloud, consumers become dependent (i.e. "locked-in") on this vendor's services and are unable to switch to a different vendor - due to the high switching costs.

1.3 FileFarm Overview

This thesis focuses on resolving the problems caused by dependence on a single storage provider. To be specific, we propose FileFarm, a cloud-of-clouds storage solution that leverages reliability of public cloud storage services while keeping needs from following aspects:

1. Preservation of data security and privacy
2. No reliance on any single cloud
3. On-demand flexibility of switching between clouds
4. Cost efficiency

Provider	Storage (\$USD/GB/Month)	Download (\$USD/GB)
Amazon S3	0.021	0.05
Microsoft Azure	0.018	0.05
Google Cloud	0.020	0.08

Table 1.1: Major pricing schema of 3 popular cloud storage providers

In FileFarm, storage nodes named "*farmers*" coordinate with each other to constitute a peer-to-peer storage network, in a sense that service availability has no dependence on any single farmer. Thus, there is no single-point-of-failure in the FileFarm storage network. With respect to the working flow, each farmer in FileFarm is linked with exactly 1 storage provider (either a public cloud or a private data server) and runs an independent server that provides storage services in terms of GET, PUT, DELETE APIs. To access FileFarm, an authenticated client establishes a stateless connection with any 1 farmer. Since the services provided by farmers are identical, a client can upload a data chunk to FileFarm through one farmer and download it later through another. FileFarm takes care of redundancy automatically, with redundant copies of data distributed over the network. When a farmer encounters service failure, an additional copy of each data chunk it stored will be stored by another farmer automatically. In case of a new farmer joining the network, storage load on each farmer will be balanced automatically.





Chapter 2

Related Work

This chapter surveys previous work in the domain of cloud-of-clouds and distinguishes FileFarm from these efforts. To preserve data confidentiality and avoid service availability failure caused by dependence on a single cloud, research related to cloud-of-clouds, or in other words, "multi-clouds" or "interclouds", has emerged and received increasing attention. Generally speaking, the main goal of cloud-of-clouds research is to seek a reliable and cost-efficient way to disperse data across multiple cloud providers and avoid dependence or data leakage on any of them. Indeed, an adequate cloud-of-clouds design needs to take a wide variety of aspects into consideration. To make a clear comparison among these efforts, we arrange this chapter into several sections, each discussing an important aspect to be considered and the corresponding mechanisms employed by each work. In the end of each section, we demonstrate the approach adopted by FileFarm and explain the differences between FileFarm and other cloud-of-clouds works.

Title	Redundancy and Confidentiality			Location Query			Storage Repair	Dynamic Load Balancing
	RAID / Erasure Coding	Secret Sharing / IDA	Other Coding	Kept by Client	Shared Metastore (DB)	Dynamical Lookup		
MCDB		✓ (Data)			✓			
DepSky		✓ (Data)			✓			
ICStore	✓	✓ (Key)			✓			
RACS	✓				✓			
HAIL	✓	✓ (Data)		✓			✓ (Client-triggered)	
CDStore	✓		Convergent Dispersal	✓			✓ (Client-triggered)	
NCCloud			Regenerating Code	✓			✓ (Client-triggered)	
Hybris	✓				✓			
FileFarm		✓ (Data)				✓	✓ (Automatic)	✓

Table 2.1: Comparison of properties provided by related cloud-of-clouds designs

2.1 Redundancy and Confidentiality



To avoid dependence of data retrievability on a single cloud, redundancy mechanisms need to be introduced into design of cloud-of-cloud systems. To provide redundancy, RACS[14] and HAIL[18] claim their approaches as RAID-like techniques used by disks and file systems, but at the cloud service level. By striping data across multiple providers, RAID-like approach helps customers to avoid vendor lock-in, reduce the cost of switching providers, and better tolerate provider outages or failures. The same benefits can also be provided by erasure coding, which is implemented in ICStore[19], CDStore[24] and Hybris[21]. In fact, RAID-like and erasure coding approaches do not differ too much from each other in the cloud context, considering the fact that cloud storage services usually provide simple key/value store APIs but not a fully functional disk; thus the RAID-like approach does not literally work like an array of pre-allocated storage spaces. Instead, it is more reasonable to view it as an array of redundant data chunks stored on different clouds, which is in the same sense as the erasure coding mechanism.

Besides erasure coding, secret sharing mechanisms such as Shamir's Secret Sharing[30] and Information Dispersal Algorithm[27] also provides redundancy as their side benefit. These mechanisms splits a piece of data into several, say n , encrypted chunks, with the assurance that a certain number $m \leq n$ of these encrypted chunks are sufficient for recovering the original data. Thus, these secret sharing mechanisms provide advantages in both redundancy and confidentiality. The mechanisms are employed by MCDB[16], DepSky[17], ICStore[19], HAIL[18], and also FileFarm.

There are still some other coding mechanisms designed to provide redundancy and confidentiality for cloud-of-cloud storage. For instance, CDStore[24] is built on an augmented secret sharing schema called convergent dispersal, which provides benefits of deduplication, storage savings and robustness against side-channel attacks. NCCloud[22] use functional minimum storage regenerating code (F-MSR) to achieve cost-effective repair for a permanent single-cloud failure.

2.2 Location Query

In a cloud-of-clouds storage system, each data chunk is distributed over some (but not all) of the clouds. To store files and retrieve them back successfully and efficiently, the system must define a way for clients to query which clouds their files are stored on, which we call *location query*. Systems like HAIL[18], CDStore[24], NCCloud[22] focus more on retrievability and algorithms of dispersal while emphasizing less on this part. From these works, we can only inferred that location information is either kept by clients or managed by other layers that are integrated with these systems.

Systems like MCDB[16], DepSky[17], ICStore[19], RACS[14], Hybris[21] address another solution to location query. In these systems, location information is stored on a shared *metastore*, which is implemented in the form of a database or a simple key/value store. By applying the metastore design, clients in these systems can efficiently get acknowledged of the clouds from which they can download data, within one or few database queries. This provides a convenient and strait-forward way for clients to access their data. Besides, since meta information is stored separately from data, the risk of malicious insider threats on clouds can be reduced significantly. For instance, Hybris[21] advocates a hybrid cloud structure in which meta data are stored on trusted premises of private servers while encrypted data chunks are stored on untrusted public clouds. With such trust-boundary design, Hybris leverages strong consistency of meta data stored off-clouds to mask the weak consistency of data stored in clouds.

Nevertheless, there are several issues related to such metastore designs. First, since the mapping between files and clouds is saved as static database records, the records will become out-of-date if some clouds encounter service outage, which has a direct impact on retrievability of files. Second, the static approach has a lack of mechanisms for dynamical replication of shard when encountering permanent single-cloud failures. As a result, the systems have low tolerance for less-reliable providers or successive changes of clouds over a long period of time. Third, considering the cases of new clouds joining the system, it takes high cost to re-distribute the data over all clouds; thus these systems would suffer from load-balancing issues and high vendor-switching cost inevitably. Last but not the

least, the metastore is regarded as a logically-centralized layer. In spite of the fact that this layer might be implemented with Byzantine Fault Tolerance algorithms or distributed synchronization tools (e.g. Apache ZooKeeper[4]), the system cannot work without this layer, which means any fault occurring to this layer will pose an direct impact on the whole system.

To solve the problems caused by a logically-centralized metastore layer, FileFarm adopts a distributed approach for location query. In FileFarm, clouds coordinate with each other to form a peer-to-peer storage network, where data chunks can be located with an efficient and dynamical lookup procedure defined by Kademlia[25] DHT protocol. Without a centralized layer storing static information, location query mechanism in FileFarm does not suffer from out-of-date issues and is robust to churn of clouds and network changes in terms of topology or scale. This is the fundamental difference between FileFarm and traditional cloud-of-clouds approaches. Based on this architectural difference, FileFarm is able to carry out more features that can not be handled well by centralized or hierarchical solutions, such as automatic storage repair, load-balancing, and better degree of fault tolerance.

2.3 Storage Repair

An important property of cloud-of-clouds systems is disaster recovery. To be specific, when a cloud encounters service failure, the redundancy of each data chunk it used to stored is reduced by a certain factor. To maintain consistent level of redundancy, a *storage repair* mechanism needs to be included in the cloud-of-clouds system design. For an erasure coding based system, an instinctive approach toward storage repair is to let clients download sufficient chunks of data back, reconstruct the lost chunk, and then re-distribute it to other clouds. This is the exact approach implemented by HAIL[18] and CDStore[24]. In fact, this is a general procedure of storage repair that can be employed to all coding-based redundancy schemas, while not explicitly described by some of the related implementations. Besides the general procedure described above, some systems meet requirements of storage repair with different coding mechanism, seeking to reduce

the repair traffic. For instance, NCCloud[22] uses network coding based schema to maintain the same data redundancy level as erasure codes, but uses less repair traffic.

While the download-and-redistribute approach manages to repair storage failure and preserve consistent level of redundancy, it is considered to be unfeasible for one primary reason: the mechanism requires *clients* to detect and trigger the entire storage repair procedure manually, which violates the design pattern of cloud services where service reliability should be maintained by the service provider internally and should not rely on any action of clients.

To address this issue, FileFarm adopts a different approach toward storage repair by leveraging its distributed nature. In FileFarm, each stored chunk of data is "re-published" by exactly one storage node in a period of time. As public clouds are considered to be robust, the period can be set long, say, one month. When a cloud encounters service failure, the republish mechanism will repair redundancy automatically within the following one period, without any involvement of other parties required. The same mechanism also offers benefits of load-balancing when new clouds join the network, which will be explained in 4.7.1. The automatic load-balancing and storage repair properties also make a clear distinction of FileFarm from other cloud-of-clouds works.





Chapter 3

Background

In this chapter, we review important properties of several technologies or terms that are used in FileFarm.

3.1 Cloud Computing

Cloud computing is a model describing the relationship between computing resources, service providers and consumers over network connections. In this model, service providers make efforts to offer consumers a convenient, reliable and on-demand way of accessing computing resources. The providers' computing resources are normally pooled to serve multiple consumers using a multi-tenant way with different physical and virtual resources dynamically assigned and reassigned according to consumer's demand. From consumer's point of view, the capabilities available often appear to be unlimited and can be elastically provisioned and released in any amount as long as they need. Furthermore, the resource usage can usually be measured with certain metrics (e.g., storage, processing, bandwidth and active user accounts) and be reported to both provider and consumers transparently. Based on a signed contract, provider then charges consumers in regular basis according to resource usage.[26]

With cloud computing technology, enterprises obtain an easy way toward computing outsourcing, in a sense that they no longer need to commit a large amount of capital on hardware and software to build their own IT infrastructure before product launch. In-

stead, they can rapidly allocate just-enough computing power and storage space as they need with minimal management overhead. They can also elastically expand or reduce the amount of allocated resources based on changes of business scale. Due to the flexibility and reliability, cloud computing has given rise to a paradigm shift in how computing services are deployed and delivered.[28]



3.2 Cloud Storage Service

Among all types of cloud computing services, storage is nearly the most fundamental one that many others are built upon. To offer a reliable, widely-available cloud storage service, providers build infrastructure and data centers spanning all over the world. Besides hardware deployment, providers also develop software to take care of aspects such as access management, resource routing, replication schema, error correction code, load-balancing... All efforts done by providers contribute to a single purpose: to provide consumer with a robust, easy-to-access online disk space. From consumer's point of view, the storage service seems much simpler. To access the storage service, consumers connect to provider's servers using a pre-defined API. The allowed API operations should at least include POST, GET, DELETE, by which consumers can upload, download and remove data, respectively.

Cloud storage service is a mature technology and business that changes the way data are stored and accessed. Among all existing cloud storage services, Amazon AWS S3[2], Google Cloud Storage[8], Microsoft Azure[10] are 3 successful instances. Depending on usage size, access rate and service-level agreement, a storage provider may provide various storage service products. The pricing schema of cloud storage services usually involves 2 major usage factors: (1) *static storage* (2) *data transfer*. The former one is the fee of storing data statically, charged in per GB/month basis, whereas the latter one charges consumers every GB of data downloaded from the cloud (upload traffics are usually for free). The 2 factors collectively account for the primary part of storage fee.

Cloud storage services have successfully relieve individuals or enterprise consumers from affording the high cost of maintaining their own storage systems. Due to the economies

of scale, cost of cloud storage solution is usually much lower than that of building a storage infrastructure with same level of reliability on consumers' own. Thus, cloud storage has gradually dominated the choice of enterprises over other storage options[32].



3.3 Cloud-of-Clouds

Although cloud computing provides benefits in terms of low cost, elasticity and reliability, ensuring security of data stored on clouds remains an unsolved problem, as consumers often store critical and sensitive data on the services offered by a third-party provider which may be untrusted. To solve this issue, researchers have moved forward to a new research domain[15]. "Cloud-of-clouds", or "multi-clouds", "interclouds" is an area of research aiming to build a service on multiple clouds and avoid dependence or data leakage on any of them. The term was firstly introduced by Vukolic[33]. Multiple designs have been proposed at around the same time[18],[14],[19],[17]. The research in this area has often been formulated as a Byzantine fault tolerance problem[23], in the sense that any faults occurring to a cloud may lead to misbehavior of it, while the system is designed to tolerate certain level of concurrent cloud failures. Besides, approaches of solving the single-cloud problem often integrate certain means of coding techniques that not only add redundancy to data, but also make sure that a file is not view-able or recoverable from any single cloud. This way, risks of malicious insider or single cloud service failure can be reduced significantly.

3.4 Hybrid Cloud

Hybrid cloud is a specialized branch of cloud-of-clouds research. In the settings of hybrid cloud, part of the enterprises' service is held by their own servers. Integration of both public clouds and private servers contributes to a more rapid and robust service. Such hybrid design is actually a more practical and feasible deployment option for most enterprises. As mentioned above, public clouds bring benefits of reliability, flexibility and cost-efficiency; however, they cannot provide certain benefits of servers in private net-

work, such as security, low-latency and full control over data. With proper design, hybrid cloud systems have the potential of bringing the best of both public clouds and private premises, while minimizing the disadvantages of them. A cloud storage example under this topic is: *Hybris*[21].



3.5 Peer-to-Peer Systems

Peer-to-Peer is a computing or networking architecture in which each participant, called *node* or *peer*, shares equal responsibility of maintaining the dedicated service. In a P2P system, peers are equally-privileged and follow the same protocol to negotiate with each other, without a centralized coordinator over them. Instead of aggregating storage and computing power to a single serving machine, each peer in a P2P system contributes part of its resources to the network, and requires the resources it wants from other peers in return. A properly-designed P2P system will eventually meet all peers' need. Because of the distributed architecture, P2P model does not suffer from single-point-of-failure problem, which means such systems are generally robust to frequent churning of peers, network topology changes, or failures occurring to part of the network. Besides, since peers serve needs for each other without a centralized channel, P2P systems tend to have better throughput and service capability comparing with traditional client-server models. However, since there are no central node in a P2P network, a peer needs to coordinate with other peers on its own and achieve a consistent state of consensus with others, in terms of routing information, content location, status of other peers, ... This brings extra computational and timing overhead to peer applications. Due to this, P2P protocols are usually more difficult to design than client-server ones, and some P2P systems are suffering from efficiency and scalability issues. In spite of the difficulty, P2P systems have still found their popularity in many application domains, with content sharing being the origin of the whole concept and the most successful one so far. *Napster*[11], *BitTorrent*[5], *eMule*[6], *aMule*[3] are some successful P2P content sharing applications.

3.6 Distributed Hash Table

How to search for data over a fully-distributed network has always been a research topic drawing high attention. Of all kinds of means proposed, distributed hash table (DHT) is the most popular category due to its high efficiency. Generally speaking, DHT is a content-addressed approach in which each piece of data is assigned with a *key* defined as its hash value. The piece of data is then stored in the distributed network according to its *key*. Thus, the locations where a given piece of data should be stored are defined by its content. This introduces de-duplication feature for such systems since same content will always be stored on same set of nodes. Besides, the dispersal and randomness properties of hash function also enable DHT-based systems to be designed in a load-balancing way. In P2P applications, DHTs are often implemented as an overlay or infrastructure that more complex services can be built upon. Such applications then inherit the desired features of their underlying DHTs. A DHT protocol also defines how to lookup and retrieve data from the network. Efficiency of the lookup processes has a direct impact on performance and usability of systems based on the DHT. Most of the widely-used DHTs have logarithm-time guarantee on worst-case lookup length. Chord[31], Kademlia[25], Pastry[29], Tapestry[34] are some well-known DHT protocols, to name a few.

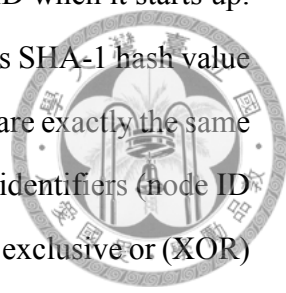


3.7 Kademlia

Kademlia[25] is a popular structured DHT protocol applied on a number of modern P2P applications including BitTorrent[5], Ethereum[7], IPFS[9] and Storj[13], ... The wide adoption of Kademlia can be attributed to several desired properties it provides. We categorize these properties into 3 aspects: *load balancing*, *efficient search* and *redundancy maintenance*.

3.7.1 Load Balancing

In a Kademlia network, storage load of composing nodes are roughly balanced. This property is assured by the underlying ID assignment and content-addressable designs. In



Kademlia, each node is assigned with a randomly-generated 160-bit ID when it starts up. Besides, each data chunk to be stored in the network is identified by its SHA-1 hash value (aka. *key*), which is also 160-bit in length. Since node IDs and keys share exactly the same format, they are also able to share the same *distance metric*. Given 2 identifiers (node ID or key), x and y , the distance between them is defined as their bit-wise exclusive or (XOR) interpreted as an integer, $\delta(x, y) = x \oplus y$. With distance metric defined, Kademlia further regulates that each piece of data should be stored at the K nodes with ID closest to the its key, where K is a system-wise redundancy parameter. Due to the randomness of node IDs and keys, data pieces are uniformly distributed over the storage nodes, with each node's load expected to be balanced.

3.7.2 Efficient Search

Kademlia provides an efficient way to search for a piece of data in a P2P network given its *key*. Like many other DHT alternatives, it takes no more than $\lceil \log(n) \rceil + c$ iterated queries to lookup any target in a Kademlia network. In this subsection, we explain the lookup mechanisms of Kademlia in detail.

As described in 3.7.1, a piece of data should be stored on the K nodes with ID closest to the its key. These K nodes are uniquely defined by the XOR distance metric, and can be found by lookup procedures specified in Kademlia protocol. To know how these procedures work, we first take a look at how routing information is kept on each node. Kademlia's ID space can be represented as a binary tree (also called *trie*) of height 160, with each ID representing a leaf node in the trie. Figure 3.1 shows an example of the trie in a small Kademlia network where ID length $d = 4$. The leaves of this trie represent all possible node IDs in this network. Given an ID x , the trie can be partitioned into d subtrees as follows:

$$D_i(x) = \{y : \delta(x, y) \in [2^{i-1}, 2^i], i = 1, 2, \dots, d\}$$

Notice that IDs in each subtree share the same prefix, where the last bit of the prefix indicates the most significant difference bit between x and IDs in the corresponding subtree, which determines the distance from these IDs to x . Thus, the longer the prefix is, the

closer the IDs are to x . For instance, IDs in $D_2(x)$ share prefix 101, which indicates that all IDs in $D_2(x)$ share the same first 2 bits with x while their third bit is different from x . Thus, distance between x and IDs in $D_2(x)$ ranges from 2^1 to $2^2 - 1$. From this point of view, it is clear that this partition splits the entire ID space into d disjoint parts according to the XOR distance metric.

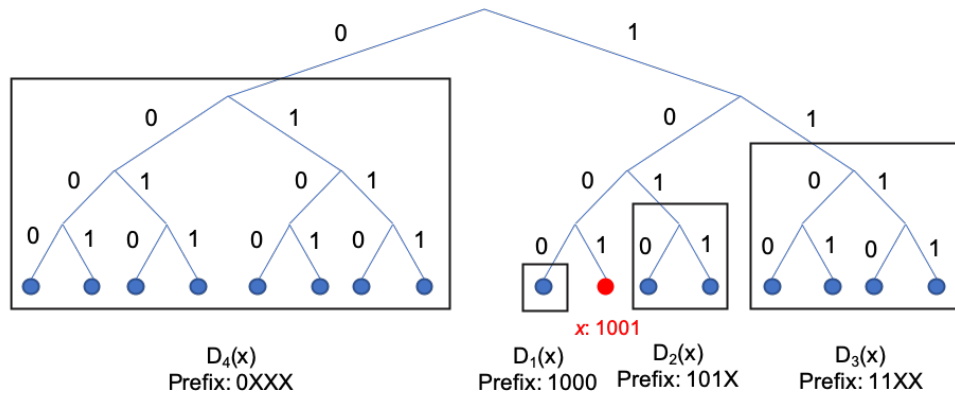


Figure 3.1: An example of Kademlia's Trie with ID length = 4

In Kademlia, a node with ID x maintains a routing table that keeps at most K records of peers for each subtrees in $D_1(x)$, $D_2(x)$, ..., $D_d(x)$. With this routing table, the node is acknowledged of peers' contact information from a wide range of distance.

Before explaining details of lookup procedures, we define 2 types of RPCs (Remote Procedure Call) used by Kademlia nodes to communicate with each other:

- *FIND_NODE*:
 - **Input**: a 160-bit TARGET_ID
 - **Output**: K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ known by the receiver with Node IDs closest to TARGET_ID
- *FIND_VALUE*
 - **Input**: a 160-bit TARGET_KEY
 - **Output**:
 - * The data chunk with key being TARGET_KEY; if the receiver does store the data chunk with key being TARGET_KEY

- * K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ known by the receiver which are closest to $TARGET_KEY$; otherwise

Now we describe the lookup procedures. In Kademlia, there are 2 types of lookup:

- *NODE_LOOKUP*: to find the K nodes in the network with ID closest to a target ID
- *VALUE_LOOKUP*: to find the content corresponding to a target key

Algorithm *NODE_LOOKUP*:

- **Input:** a 160-bit $TARGET_ID$, K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ known by the initializer with Node IDs closest to $TARGET_ID$
 - **Output:** K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ in the network with Node IDs closest to $TARGET_ID$
- The initializer puts the K closest nodes it knows into a lookup-memory.
 - The initializer picks α nodes from the lookup-memory, where α is a system-wide concurrency parameter, such as 3.
 - The initializer sends parallel, asynchronous $FIND_NODE$ RPCs to these α nodes and expect to receive K closest node records known by each of them.
 - When receiving a $FIND_NODE$ response containing K node candidates, the initializer updates these candidates to the lookup-memory and leaves only K closest ones. Then the initializer picks α un-queried nodes from the lookup-memory and sends $FIND_NODE$ RPCs to them.
 - If a round of $FIND_NODE$ fails to return closer nodes than the ones in the lookup-memory, the initializer sends $FIND_NODE$ s to all un-queried nodes in the lookup-memory.
 - The *NODE_LOOKUP* procedure terminates when the initializer receives all responses. Now the K nodes in lookup-memory are the K closest ones to $TARGET_ID$ over the network and should be returned.

Algorithm *VALUE_LOOKUP*:

- **Input:** a 160-bit *TARGET_KEY*, K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ known by the initializer with Node IDs closest to *TARGET_KEY**

- **Output:**

- The *TARGET_CHUNK* with key being *TARGET_KEY*; if *TARGET_CHUNK* exists in the network
 - K triples of $\langle IP\ address, UDP\ port, Node\ ID \rangle$ in the network which are closest to *TARGET_KEY*; otherwise
- (i) The initializer puts the K closest nodes it knows into a lookup-memory.
 - (ii) The initializer picks α nodes from the lookup-memory, where α is a system-wide concurrency parameter, such as 3.
 - (iii) The initializer sends parallel, asynchronous *FIND_VALUE* RPCs to these α nodes.
 - (iv) When receiving a *FIND_VALUE* response containing *TARGET_CHUNK*, the procedure ends by returning *TARGET_CHUNK*.
 - (v) When receiving a *FIND_VALUE* response containing K node candidates, the initializer updates these candidates to the lookup-memory and leaves only K closest ones. Then the initializer picks α un-queried nodes from the lookup memory and sends *FIND_VALUE* RPCs to them.
 - (vi) If a round of *FIND_VALUE* fails to return closer nodes than the ones in the lookup-memory, the initializer sends *FIND_VALUES* to all un-queried nodes in the lookup-memory.
 - (vii) The *VALUE_LOOKUP* procedure terminates when the initializer receives all responses. In this case, the initializer fails to find *TARGET_CHUNK*, and the K nodes in the lookup memory closest to *TARGET_KEY* should be returned.



Now we want to discuss the efficiency of NODE_LOOKUP and VALUE_LOOKUP procedures. According to the sketch of proof in [25], NODE_LOOKUP in a Kademlia network will finish in $\lceil \log(n) \rceil + c$ steps for some small constant of c , where n is network size, i.e., number of nodes in the network. This gives an rough upper bound for NODE_LOOKUP efficiency.

Different from NODE_LOOKUP, the VALUE_LOOKUP procedure finishes immediately when the target value is found. Thus, VALUE_LOOKUP procedure only needs to reach any one of the K closest nodes instead of finding all of them. According to Cai's analysis[20], it takes no more than $(1 + O(1)) \frac{\log(n)}{H_K}$ steps for any node in a Kademlia network to locate any other node, where $H_K = \sum_{i=1}^K 1/i$. This upper bound also stands for VALUE_LOOKUP, considering the fact that VALUE_LOOKUP for the target key converges along the same path as NODE_LOOKUP for the closest farmer, due to *unidirectionality* of XOR distance metric: For any given point x and distance $\Delta > 0$, there is exactly one point y such that $d(x, y) = \Delta$.

From the analysis above, we know that both NODE_LOOKUP and VALUE_LOOKUP procedures are guaranteed to be finished in logarithm steps, while VALUE_LOOKUP takes a tighter bond with a factor of $H_K = \sum_{i=1}^K 1/i$, since only 1 copy of data is required to be found, and we can find it by reaching any of the K storing nodes.

3.7.3 Redundancy Maintenance

Redundancy maintenance is another desired property provided by Kademlia. When a node churns off from the network, each of the data chunks it used to stored will lose one redundancy copy. Kademlia has a *efficient key republishing* mechanism ensuring that each chunk will always have at least K copies over the storage network.

The *efficient key republishing* mechanism regularly checks if each data chunk is stored on the K closest farmers. If any of the K closest farmers does not store the chunk, a copy will be sent to it automatically. This assurance is provided by the design in which each farmer periodically attempts to *republish* each chunk it stored to the other $K - 1$ farmers who should store the same chunk. During the republishing period, if a farmer receives the

republishing message of a chunk from other farmer, it assumes the message has also been sent to the other $K - 1$ closest farmers and thus it should not republish it again during this period in order to reduce traffic. As long as the republishing interval of farmers are not exactly synchronized, there will only be exactly 1 republishing message for each chunk in each interval. However, the efficient key republishing mechanism has a considerable overhead. Within a *republishing period*, each chunk induces one NODE_LOOKUP followed by K republishing messages. Choices of longer republishing period would mitigate the overhead while providing weaker guarantee on retrievability of files.

3.8 Information Dispersal Algorithm

Information Dispersal Algorithm (IDA)[27] is a computation schema of dispersing content of a file into smaller chunks and reconstructing the original file based on some of the chunks, where the dispersal and reconstruction process are computationally efficient. To be more precise, a (p, q) IDA schema breaks a file F into $p + q$ chunks of size $\frac{|F|}{p}$, such that F can be reconstructed from any p chunks but not less. Figure 3.2 shows an example of dispersing a file F of size $|F| = 16$ bytes with a $(4, 2)$ schema. In the figure, each byte is represented as an integer from $[0, 255]$. The first step of IDA is to split the original file into $p = 4$ chunks, with each chunk being a row of the matrix representing the entire file. After that, a randomly generated matrix M of size $(p + q) \times p$ is being multiplied with the file matrix and the computation yields a matrix of size $(p + q) \times \frac{|F|}{p}$. Each row in the result matrix is a computed chunk that is supposed to be stored in a place different from others. To reconstruct the original file F , one needs to collect p computed chunks and multiply the collected chunks with inverse matrix of concatenation of their corresponding rows in M . As long as the corresponding rows in M are linearly independent, the original file can be reconstructed based on these p collected chunks.

Due to its potential traits of security, load-balancing and fault tolerance design, Information Dispersal Algorithm is widely-used in distributed file storage and content sharing applications. Considering a distributed network of computers and workstations, files pre-processed with IDA are certainly hard to be reconstructed by nodes other than the uploader,

since only the uploading node has the knowledge of dispersion keys and distribution of chunks in the network. Besides, in a distributed application, the stored chunks can be downloaded from different nodes in parallel, this boosts the downloading performance of large files. Also, the q factor in the IDA schema makes redundancy and tolerance in a fault-prone environment. These are reasons why IDA is suitable for distributed systems.

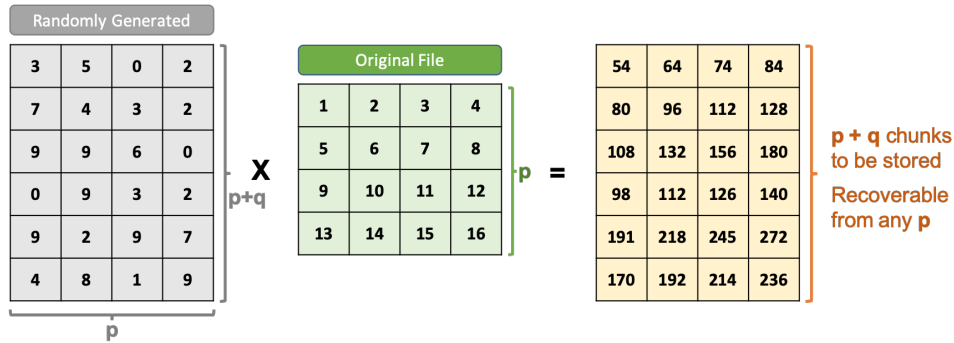


Figure 3.2: An example of IDA with schema $(p, q) = (4, 2)$

3.9 Public Key Infrastructure

Public key infrastructure (PKI) is an infrastructure that enables entities to securely communicate over an insecure public network via cryptographic techniques of public-key encryption, digital signature and certificate-based authentication. From the perspective of cryptography, PKI is a reliable approach to bind public keys with respective identities. The binding process is facilitated by registration and issuance of digital certificates, which can be demonstrated by Figure 3.3: The whole process involves 3 parties: an entity providing service to users, an certificate authority (CA) and a user. To get a certificate, the entity needs to generate a certificate request (CSR) containing its public key first. It then sends the CSR to CA. With all means of verification, CA finally trusts the entity. CA then generates a certificate for the entity, which includes CA's digital signature signed with its private key. The certificate is then sent back to the entity and installed on its server machine. From now on, whenever a user sends a connection request to the entity, it responds with its certificate. The user then verifies this certificate by validating the CA's signature on the certificate. If the verification passes, the user will establish a secure connection

with the entity encrypted by the entity's public key.

PKI is a general term involving policies of issuing, managing, and validating digital certificates. Inside this broad term, X.509[12] is a majorly adopted standard defining the format of public key certificates, which is widely used in the Internet and is the fundamental technique implemented in TLS/SSL, the basis of HTTPS.

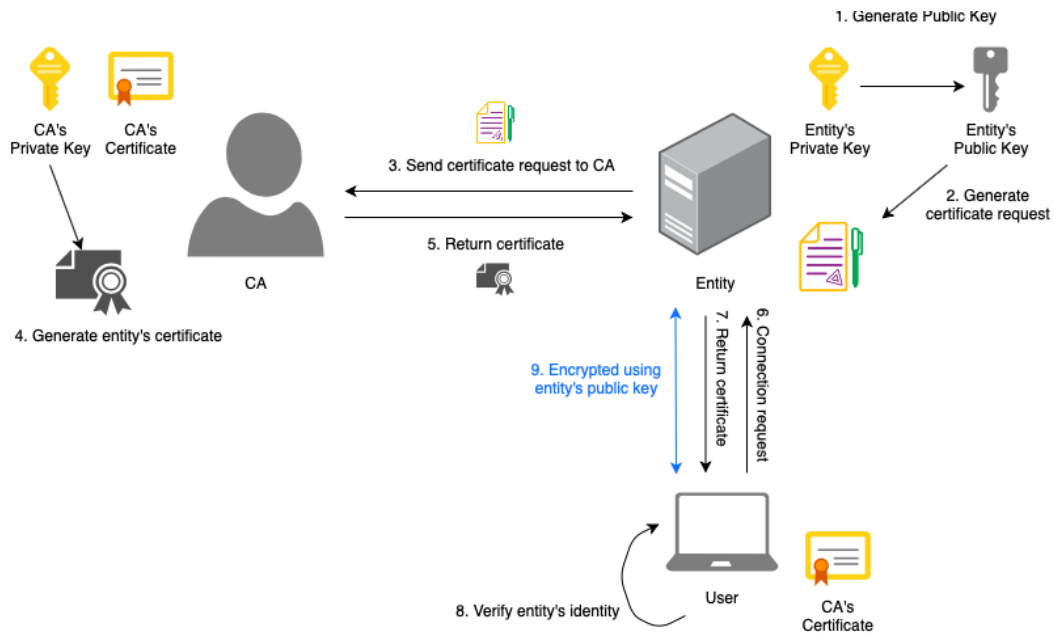


Figure 3.3: Procedure of certificate issuance and certificate-based authentication





Chapter 4

Methodology

This chapter describes FileFarm system in detail. It starts by presenting the system architecture and roles in the system. Then it shows the model of each building block of FileFarm and the upload / download process flows. Finally it explains each of the desired characteristics FileFarm provides and the mechanisms behind them.

4.1 System Architecture

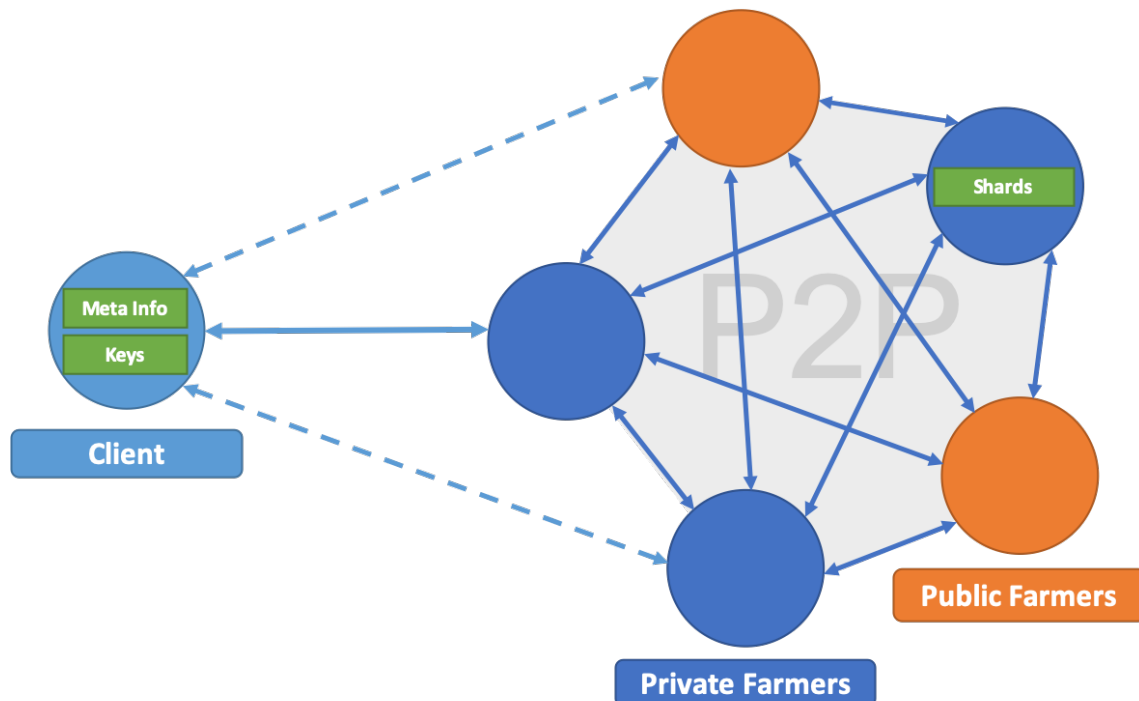


Figure 4.1: System architecture

The FileFarm system is composed of 2 roles:

1. *Client*: User-side application providing user interface, handling upload/download tasks and managing file meta along with hash values and encryption keys for later retrieval.
2. *Farmer*: Basic component of the FileFarm storage network. A farmer is linked with exactly 1 storage provider (either a public cloud or a private data server) to offer storage service. At the same time, farmers connect to each other to form a P2P network and take care of availability of each stored piece of data. A farmer also provides API for clients to store and retrieve data.



As shown in Figure 4.1, the FileFarm cloud-of-clouds is formed by farmers, in a sense that service availability has no dependence on any single farmer or its underlying cloud. Thus, FileFarm system has no single-point-of-failure. All farmers provide the same storage service. Thus, a client can establish stateless connection with any farmer for uploading/downloading data.

The basic unit of data in FileFarm network is called *shard*, which is a computed segmentation of file that is saved in the network. Depending on the *sharding schema*, a specific amount of distinct shards is required to reconstruct the original file. From client's point of view, a file is split, encrypted and then computed into a number of shards with IDA (more details explained in 4.5). Instead of the original file, the shards are what actually uploaded to farmers. The encryption keys, hashes, file meta are kept by the client itself. This ensures that only the client itself is able to reconstruct its own data.

4.2 Application Models and Process Flows

Figure 4.2 shows the model of client and farmer applications. Client application has a interface for user to manage their own storage and to upload/download files. When receiving command from user to upload a file, the client application splits the file into slices and pre-process each slice with encryption engine and IDA engine. After pre-processing,

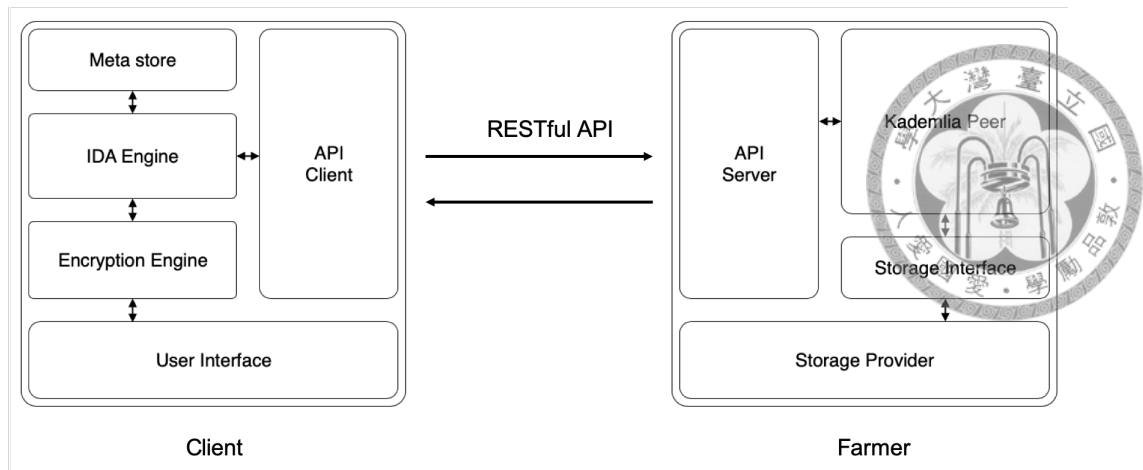


Figure 4.2: Application model of client and farmer

each slice is transformed into multiple shards, and the encryption keys and hash value of shards are saved in the client's meta store at the same time. The client application then uploads each shard to the FileFarm storage network through a randomly-chosen farmer. Farmer application, on the other hand, has an API server module that serves request from client applications. Besides the server module, farmer also has a Kademlia peer module that coordinates with other farmer's corresponding unit to synchronize storage status of uploaded shards and provide an efficient way to locate them. In order to apply farmer application on different storage services, FileFarm implements an abstract storage interface layer that provides identical out-bounded API but negotiates with different storage providers according to their API format.

Going with the application model, we present upload / download flow of FileFarm in figure 4.3 and figure 4.4, respectively. Some of the processes in these 2 flow diagrams is not clear yet, which will be explained in the following sections.

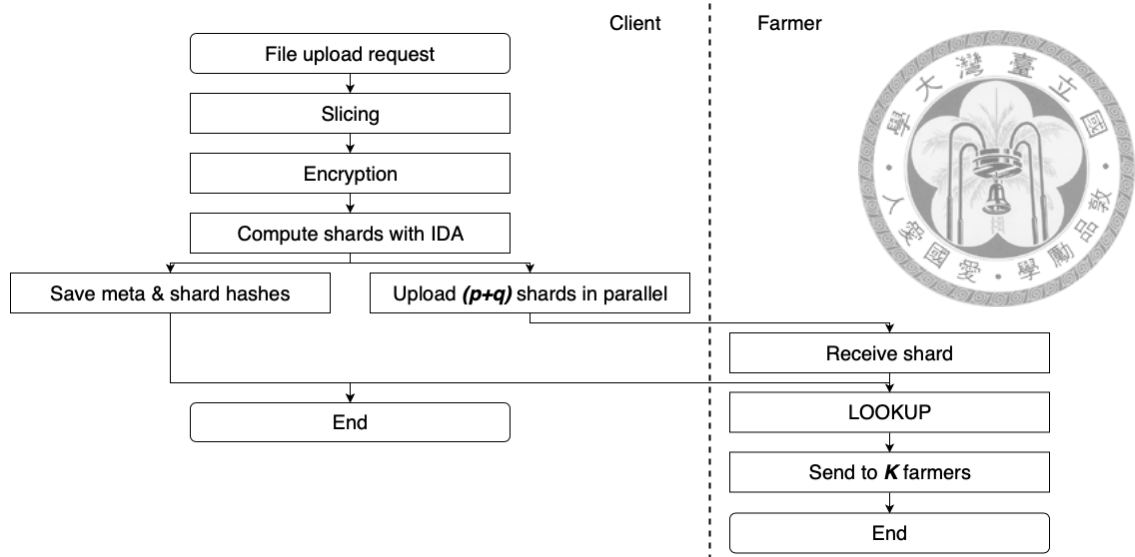


Figure 4.3: Upload flow

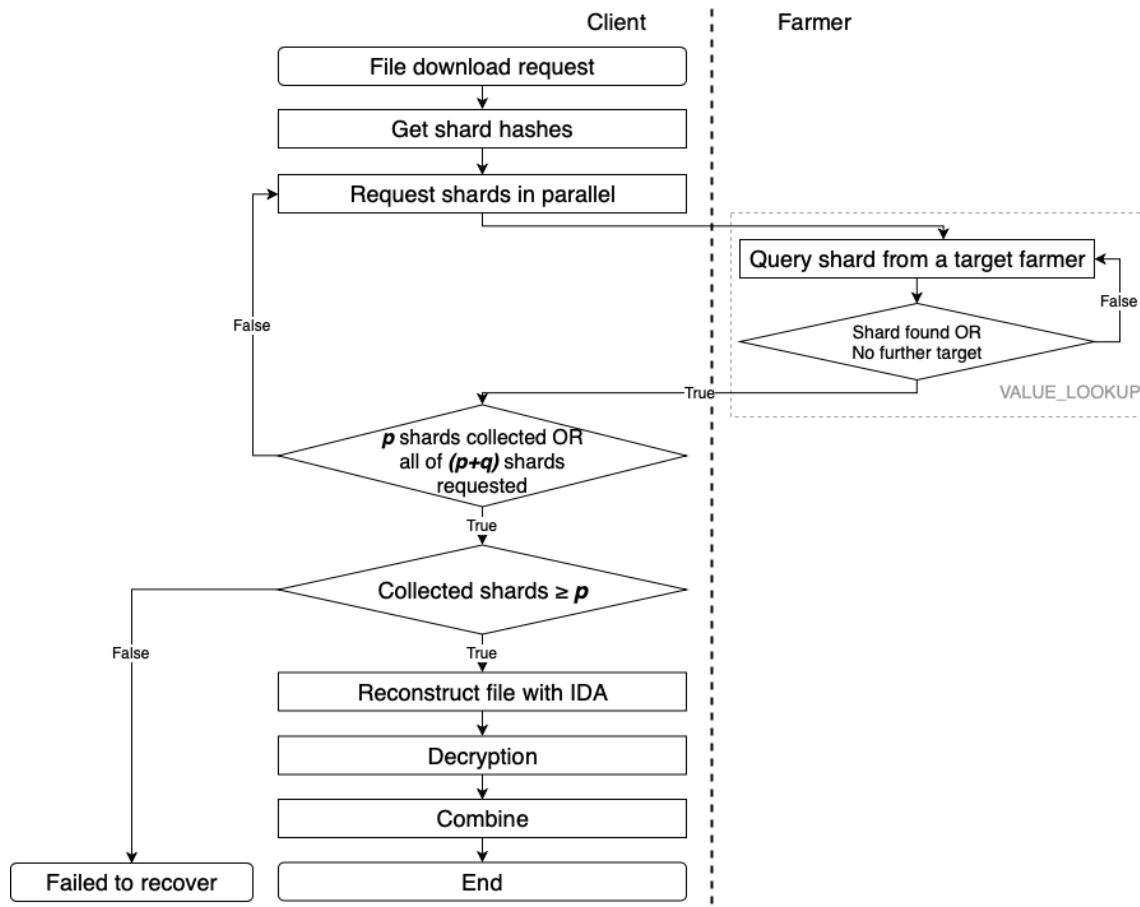


Figure 4.4: Download flow

4.3 DHT-Based Approach

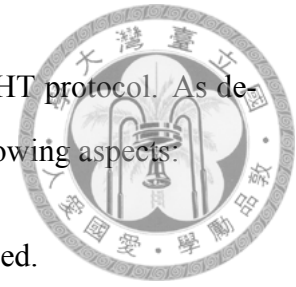
To find shards in a P2P network, FileFarm implements Kademlia DHT protocol. As described in 3.7, Kademlia protocol gives FileFarm benefits in the following aspects:

1. *Load balancing*: storage load of each farmer is roughly balanced.
2. *Efficient search*: lookups in FileFarm need no more than logarithm steps.
3. *Redundancy maintenance*: each shard is always stored on at least K farmers.

Instead of saving location of shards as static records in a centralized database, FileFarm adopts Kademlia's dynamical lookup procedures described in 3.7.2. Before storing a shard, a farmer uses the `NODE_LOOKUP` procedure to determine on which farmers the shard should be stored, and then sends parallel storing messages to these farmers. In the case of shard retrieval (download), a farmer performs `VALUE_LOOKUP` procedure to find the shard iteratively until it is found. Both `NODE_LOOKUP` and `VALUE_LOOKUP` procedures are robust to churn of farmers, network topology changes and scaling of network size, which usually can not be handled well in centralized solutions. Besides, since given shard, the K farmers to store it are uniquely defined and the results of `NODE_LOOKUP` and `VALUE_LOOKUP` procedures have no dependence on originating node, a client can request any farmer to upload or download the shard, which will eventually be stored on the same set of farmers. This allows a parallel and fault-tolerant design in which clients can upload /download different shards from different farmers simultaneously and do not rely on a specific farmer to perform these tasks.

4.4 Beyond Kademlia

From 4.3, we acknowledge that FileFarm inherits a number of benefits from Kademlia. However, Kademlia was originally invented for content-sharing applications but not storage systems. To claim FileFarm as an enterprise storage, we also need to consider following requirements that cannot be provided by Kademlia:



1. *Data confidentiality*
2. *Access management*
3. *Cost efficiency*
4. *Retrievability*



Corresponding mechanisms are designed to meet these requirements:

1. *Encryption and Information Dispersal Algorithm*
2. *Decentralized authentication*
3. *Storage release and prioritized download*
4. *Public farmer ID assignment*

These mechanisms will be explained in the following sections 4.5, 4.6, 4.7, 4.8.

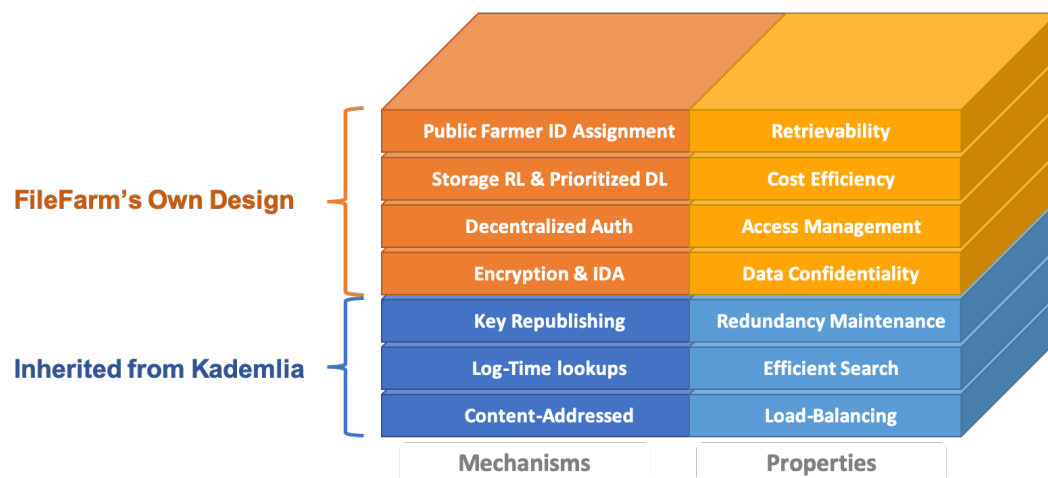


Figure 4.5: FileFarm's property stack

4.5 Data Confidentiality

Designed to be a general-purposed P2P content sharing overlay, vanilla Kademlia protocol does not provide features of encryption and dispersion since data are meant to be shared instead of being kept in secrecy in most P2P applications. Thus, without confidentiality

design, files will be uploaded to Kademlia network directly and the storing nodes will be able to peek content of the files. However, as FileFarm is a private storage system targeting on preserving data security and privacy, the mechanisms concerning data confidentiality must be designed.



In order to preserve privacy and confidentiality while storing data on public clouds, FileFarm introduces a pre-processing flow of files before they are actually uploaded, in a sense that every files are randomly dispersed over clouds, and only the owner holds the key to retrieving and reconstructing them. To be precise, an example is shown in Figure 4.6. A file F of size S is sliced into chunks $C1, C2, \dots, Cn$ of size $\frac{S}{n}$ and then chunks are encrypted into $E1, E2, \dots, En$ of the same size, respectively. For each of the encrypted chunk, a *sharding* process is performed based on an Information Dispersal Algorithm of (p, q) schema (see 3.8), which produces $p+q$ shards of size $\frac{S}{n} * \frac{1}{p}$, and the original encrypted chunk can be recovered from any p of these $p+q$ shards. Now the pre-processing flow is finished, and the $n * (p+q)$ generated shards are uploaded to FileFarm in parallel.

To reconstruct the original file F , the procedure seems inverted. For each of the encrypted chunks $E1, E2, \dots, En$ that was sharded into $p+q$ shards, the client sends parallel requests to collect p shards so that it can be reconstructed using IDA. The reconstructed chunks are then decrypted to $C1, C2, \dots, Cn$ and combined into the original file, F .

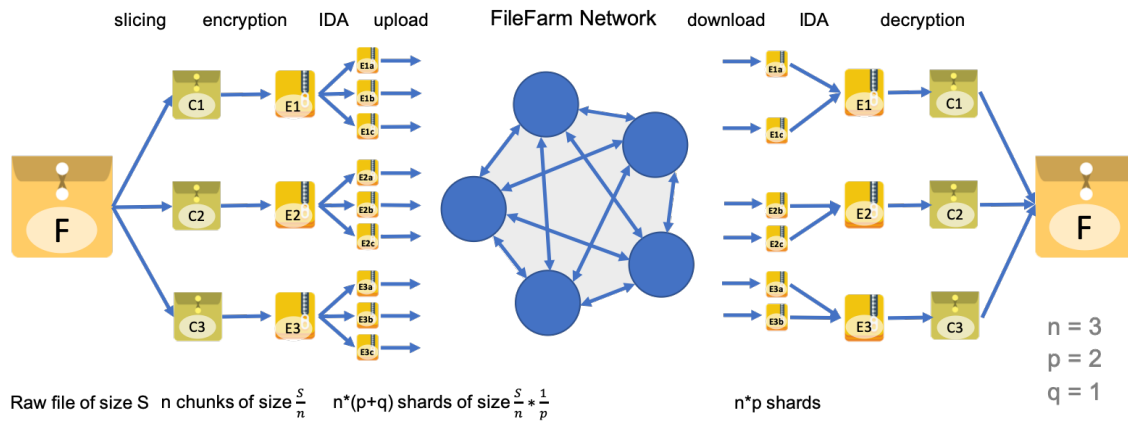


Figure 4.6: An example of encryption and sharding flow

4.6 Access Management

Different from general P2P applications, FileFarm is an enterprise storage system applied in a **private** context. In such usage scenario, how to grant access to authorized users while denying connections from unauthorized ones plays a crucial role in security of the system. To achieve this goal, FileFarm adopts a decentralized approach toward access management. Different from common web systems, the access management design in FileFarm does not involve a centralized key directory. Instead, the account and key information is saved distributed-ly in FileFarm storage network, just like normal data, which means the access management mechanism has no dependence on any single farmer, and no data leakage will occur even if any farmer is compromised.

4.6.1 Decentralized Authentication

The *decentralized authentication* mechanism involves 2 major parts: (1) *register*, (2) *login*.

1. **Register:** This process occurs when a user accesses the FileFarm system for the first time. From user's point of view, the goal of this process is to create an account, get permission from the administrator and then save account information into the network so that it can be accessed successfully next time, probably from a different client machine. This approach is based on the concept of Public Key Infrastructure, which is described in 3.9, whereas the goal here is to validate clients but not serving hosts. The approach involves 3 roles: an administrator, the FileFarm cloud, and a user accessing FileFarm through a client application. To register an account, the user determines an account name($Account_U$) and a password ($Passwd_U$) in the client application. The account name and the password will be hashed into user ID (ID_U) and private key ($PriKey_U$), respectively. The client application then checks existence of the account name by querying from the FileFarm cloud with user ID. If the account has not been registered yet, the procedure goes on. The client application will wrap (1) account name, (2) user ID, (3) user information,

(4) a public key ($PubKey_U$) generated from the user's private key and (5) a user's signature (Sig_U) into a certificate request (CSR). On receiving such request, the administrator makes a decision on whether to grant access for the client or not. If the client passes the administrator's validation, the administrator creates a signature (Sig_A) on the signing request, which turns into a certificate. The certificate is then sent back to the client. When receiving a valid certificate, the client application stores it on the FileFarm cloud with key being her user ID, and the registration process is finished successfully.

2. **Login:** This process is designed for a registered user to login to his account with account name and password, which are the only information needed to be carried by the user. The user can login via a client machine different from the one he used to register his own account. To login to FileFarm, the user first types account name ($Account_U$) and password ($Password_U$) into the client application interface. The account name and the password are hashed into user ID (ID_U) and private key ($PriKey_U$), respectively. Then the client application makes a query for the user's certificate from the FileFarm cloud. On receiving the certificate, the client application verifies the user's password by checking the signature on it. If the password validation succeeds, the client is allowed to access the FileFarm storage network. To access FileFarm, the client appends its certificate to the connection request packet, and then encrypts the payload with its own private key. When receiving request from clients, a farmer checks the validity of the certificate by verifying the signature on it using the administrator's public key. The farmer also verifies the client's identity by decrypting the payload using the client's public key specified in the certificate.

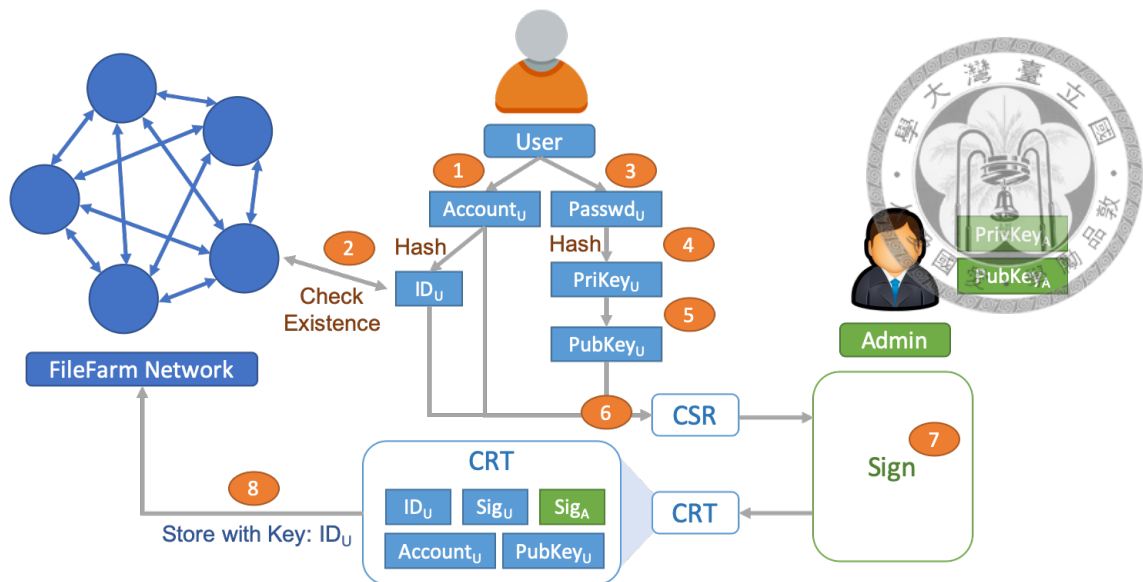


Figure 4.7: An illustration of register procedure

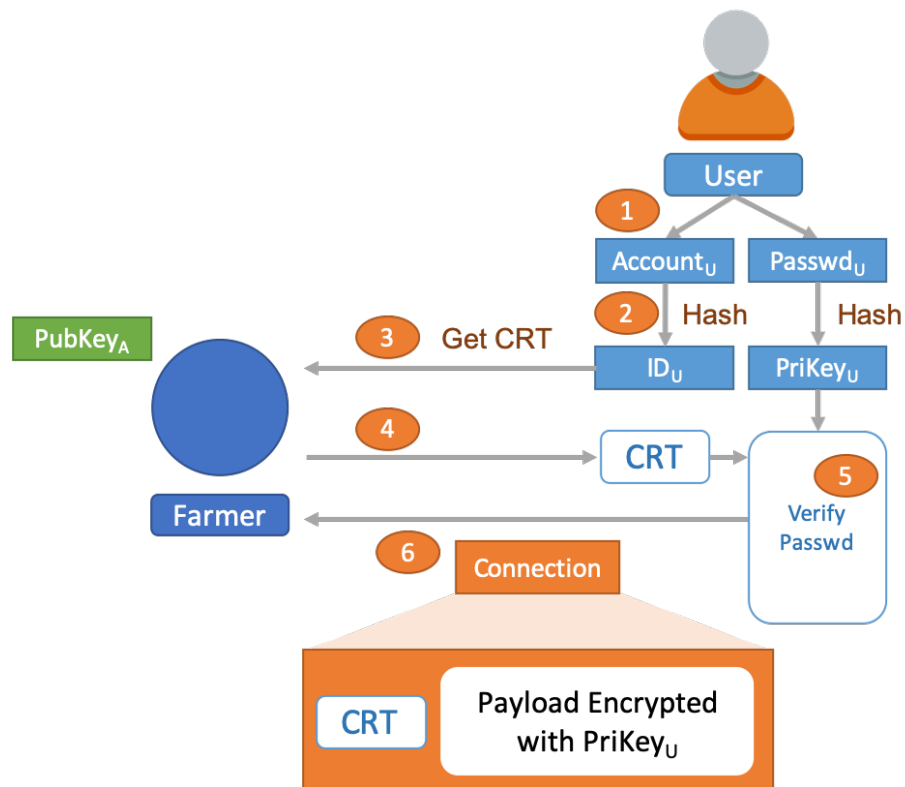


Figure 4.8: An illustration of login procedure



4.7 Cost Efficiency

FileFarm is built on the top of both public clouds and private premises, whereas storage fees are charged by the public storage providers but not by the enterprises' own servers. Due to this fact, any storage operation executed on public clouds needs to be carefully inspected, in order to minimize the fee charged by the public storage providers without lost of retrievability. As depicted in 1.2, there are 2 major kinds of storage fees: (1) *static storage fee* per-GB / per-month (2) *data transfer out fee* per-GB of download traffic. We will explain how the redundant fee comes from the original design and introduce the mechanism employed by FileFarm to reduce these fees in the following sub-sections.

4.7.1 Storage Release

The *storage release* mechanism reduces *static storage fee* by releasing unused redundant copies of shard and keeping redundancy at *exactly* K but not more. To understand why there might be unused redundant copies of shard, we need to take a look at how the underlying Kademlia DHT protocol maintains redundancy.

As described in 3.7.3, Kademlia implements an *efficient key republishing* mechanism that ensures each shard always has "at least" K copies over the storage network. However, there are certain scenarios making number of copies more than K , which results in unneeded storage overhead. Considering a scenario in which a shard is kept by K farmers and one of them churns off accidentally. With efficient key republishing, one of the remaining $K - 1$ farmers will detect this and send a copy of the shard to the $K + 1^{th}$ closest farmer, so that redundancy recovers from $K - 1$ to K . However, when the failed farmer recovers from service outage, the redundancy will raise from K to $K + 1$. The copy on the $K + 1^{th}$ closest farmer is now an unused overhead that needs to be eliminated in order to save static storage cost.

This is the moment that *storage release* mechanism should work. Due to the fact

that republishing message will only be sent to the K closest farmers, the $K + 1^{th}$ closest farmer will not receive this message, which triggers it to republish the shard in the next period. To find the K closest farmers to this shard, a `NODE_LOOKUP` procedure should be performed before sending republishing messages. This gives the $K + 1^{th}$ closest farmer a chance to find itself not being one of the K closest farmers, it then stops the republishing process and deletes the shard from its storage. This shows the entire working procedure for *storage release* mechanism.

From the explanation above, we make a brief conclusion that *storage release* is a mechanism that minimizes *static storage fee* by holding redundant copies of each shard at exactly K but not more. Besides, as the K -redundancy guideline still holds, *storage release* does not induce retrievability concerns.

4.7.2 Prioritized Download

Prioritized download is a mechanism aiming to minimize the *data transfer out* fee from public clouds. This mechanism only makes sense in hybrid cloud settings, where not only public clouds but also enterprise's private storage machines such as NAS or SAN are contributed to the storage system. Without *prioritized download* mechanism, each farmer is treated equal and shares the same probability of serving data for clients. However, the cost of downloading data from public clouds is significantly higher than that of downloading from private servers, because of the *data transfer out fee* charged by the storage providers. *Prioritized download* is the mechanism ensuring that shards are preferred to be downloaded from private farmers, so that *data transfer out fee* is minimized.

In hybrid-cloud settings, enterprises optionally build their own private farmers. The private farmers are usually not as reliable as public ones. However, they have advantages in at least 2 aspects:

1. High throughput and low delay in local area network (LAN)
2. No data transfer fee needed

These advantages make it reasonable for enterprises to split a portion of budgets on

investment of private storage devices. This not only reduces data transfer fee charged by public clouds, but also improves download performance.

To understand how prioritized download works, we need to take a closer look at how the underlying Kademia DHT protocol retrieves a shard. As described in 3.7.2, Kademia implements an iterative VALUE_LOOKUP procedure to retrieve shards. The VALUE_LOOKUP procedure finishes immediately when any of the visiting farmers does store the shard, and the client will download the shard from that farmer.

Prioritized download is a modified version of VALUE_LOOKUP, with some noticeable differences:

1. Prioritized download finishes once the target shard is found on a **private** farmer.
2. Prioritized download memorizes a public farmer who has the shard when finding such one, but not download from it immediately.
3. Prioritized download downloads from a public farmer only if there is no private farmer found storing the target shard.

By applying these differences to the VALUE_LOOKUP procedure, *prioritized download* ensures that shards will be downloaded from public farmers only if there is no private farmer serving the same shard. This effectively reduces the data traffic flowing out from public clouds and also the *data transfer out fee* charged by storage providers. Since *prioritized download* only changes the preference of downloading order, any public farmer serving the shard will eventually be chosen once there are no private farmers serving the same shard. This implies that *prioritized download* does not affect retrievability of files.

From the description above, we know that FileFarm treats public farmers and private farmers differently. In fact, a special approach is implemented by FileFarm to achieve this kind of differentiation. We will explain the details and the benefits brought by such differentiation in 4.8.1.

4.8 Retrievability

In FileFarm's design, redundancy introduced by the K factor of Kademlia (3.7) and the q factor of Information Dispersal Algorithm (3.8) both contribute to retrievability of files. However, considering a hybrid-cloud case in which some shards are stored totally on private farmers but no public ones. The significant reliability difference between public and private farmers would cause a large disparity in retrievability of files, which makes the system's behavior unpredictable. To provide a more rigorous guarantee on retrievability in hybrid-cloud settings where reliability of public clouds and private servers differs greatly, we have to make sure every shard is saved on at least 1 public cloud with high probability. This demand gives rise to the design of *public farmer ID assignment*.

4.8.1 Public Farmer ID Assignment

To differentiate public farmers from private ones, FileFarm gives public farmers a special identity. In the 160-bit farmer ID space, each public farmer is assigned with an ID with last 144 bits be '0', hence there can be at most $2^{16} = 65536$ public farmers in the network. In contrast, ID of private farmers are generated randomly but cannot have last 144 bits be '0'. Besides, public farmer IDs are issued in *bit-reversal permutation ordering*. To be clear, we explain this kind of ordering in a simplified case where public IDs vary in the first 4 bits and have last 156 bits be '0' (see figure 4.9). To generate ID for the i^{th} public farmer, we reverse the binary representation of the number $i - 1$ and assign the resulting binary string as the first 4 bits of the newly-generated ID. Following this rule, the first 4 bits of generated public IDs will be '0000', '1000', '0100', '1100', ...

With public farmer IDs being assigned in bit-reversal permutation ordering, we can be certain that public farmers are dispersed over the 160-bit ID space. Since that hash values of shards follow a uniform distribution over the 160-bit key space and that a shard is stored on farmers with ID closest to its hash value, the dispersion of public farmer IDs makes them load-balanced in terms of storage usage and request frequencies. Besides, this explicit assignment also avoids the situation that public farmer's IDs distribute uneven over the network, which would cause disparity between retrievability of shards.

To sum up, the assigning rule of public farmer ID brings following benefits:

1. Provides an explicit rule to distinguish public farmers from private ones, which is needed for prioritized download (4.7.2) to work.
2. Poses a strong guarantee on load-balancing of public farmers, which in further minimizes the reliance on any single public cloud.
3. Maximizes the likelihood of each shard to be stored by at least one public farmer, which improves the average retrievability of each shard and thus enhances the overall retrievability of files.

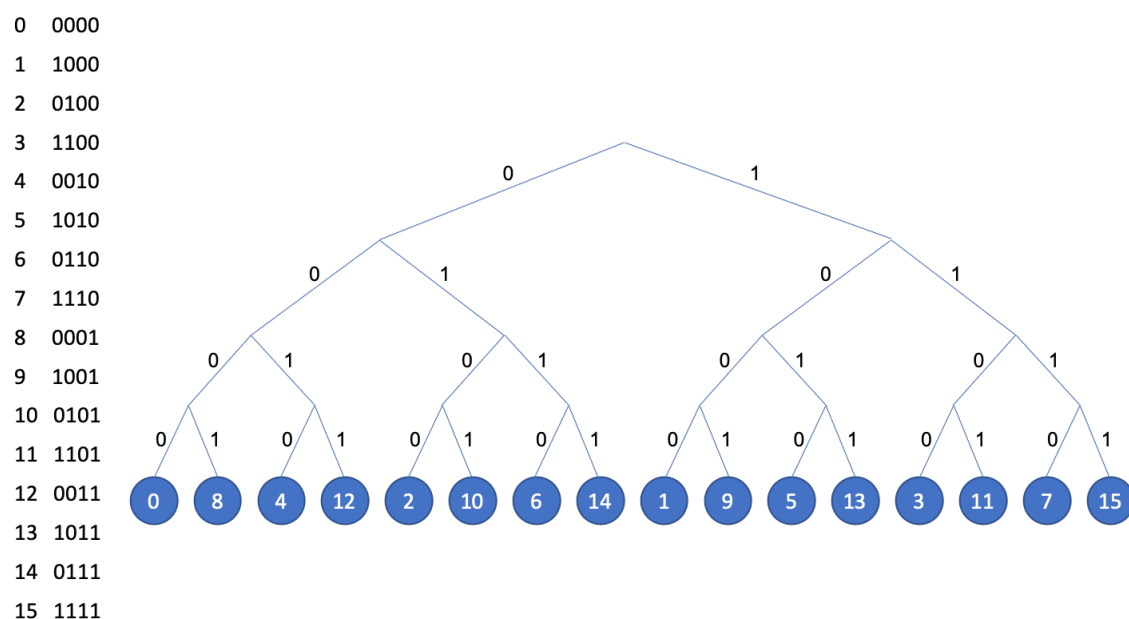


Figure 4.9: Bit-reversal permutation ordering with prefix length = 4





Chapter 5

Experiments and Results

To verify the claimed properties of FileFarm, we conduct a series of experiments. In this chapter, we will describe our experimental environment first, and then describe settings, process, and results of each experiments.

5.1 Environment

In the following experiments, we run FileFarm on 10 physical hosts with following system information:

- CPU: Intel(R) Xeon(R) CPU E5-1630 v3 @ 3.70GHz
- Memory: 16 GiB DDR4
- Network Interface: Ethernet Connection (2) I218-LM (1Gbit/s)
- Operating System: Ubuntu Server 18.04.2 LTS

Each of the 10 physical hosts is assigned with a static IP address, and all of the 10 IP addresses belong to the same subnet with mask 255.255.255.0. Depending on settings of each experiment, a physical host might runs one or multiple instances of FileFarm farmers and clients simultaneously.

5.2 Experiment: NODE_LOOKUP Efficiency

In FileFarm, each shard is stored on exactly K farmers. Instead of saving location of shards as static records in a centralized database, FileFarm adopts Kademlia's dynamical lookup procedures. Thus, the efficiency of these procedures will impact system's I/O performance greatly. According to the sketch of proof in [25], NODE_LOOKUPS in a Kademlia network will finish in $\lceil \log(n) \rceil + c$ steps for some small constant of c , where n is network size, i.e., number of nodes in the network. We want to verify that this property also holds in FileFarm.

In this experiment, we first start n farmers. After all farmers are bootstrapped, we make each farmer perform NODE_LOOKUP on 100 random targets and report the number of steps needed to locate the K closest farmers around the target. Then we collect and compute mean of steps needed. The whole process is repeated for $n = 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150$ and $K = 1, 2, 3, 4, 5$.

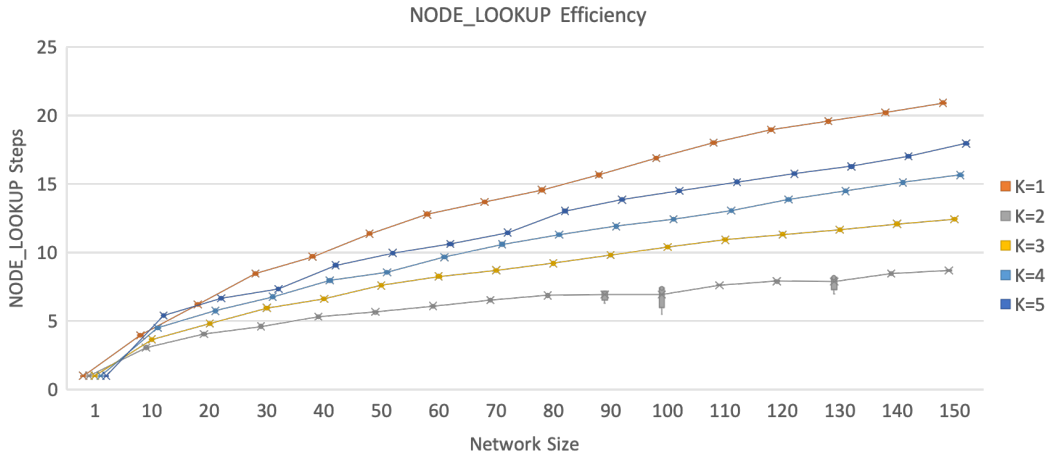


Figure 5.1: NODE_LOOKUP steps with respect to redundancy parameter K and network size n . The curve shows a logarithm relationship between NODE_LOOKUP steps and network size.

From Figure 5.1, we can observe that number of NODE_LOOKUP steps grows with number of farmers, following a logarithm-like curve. With number of farmers growing from 50 to 100, it only takes around 2 more steps to locate all K closest farmers, which makes FileFarm network scalable. However, it can also be observed that a larger setting of K requires more steps for NODE_LOOKUP procedure to finish, which is intuitive,

considering the fact that NODE_LOOKUP finds all of the K closest farmers, instead of one or some of them.



5.3 Experiment: VALUE_LOOKUP Efficiency

Just like performing NODE_LOOKUP before uploading a shard, farmers perform VALUE_LOOKUP before downloading a shard. Different from NODE_LOOKUP, the VALUE_LOOKUP procedure finishes immediately when the target value is found. Thus, VALUE_LOOKUP procedure only needs to reach any one of the K closest farmers instead of finding all of them. According to Cai's analysis[20], it takes no more than $(1 + O(1)) \frac{\log(n)}{H_K}$ steps for any node in a Kademlia network to locate any other node, where $H_K = \sum_{i=1}^K 1/i$. This upper bound also stands for VALUE_LOOKUP, considering the fact that VALUE_LOOKUP for the target key converges along the same path as NODE_LOOKUP for the closest farmer, due to *unidirectionality* of XOR distance metric. In this experiment, we want to verify this property on FileFarm and compare the result with 5.2.

In this experiment, we first start n farmers. After all farmers are bootstrapped, we start 10 clients and make them upload 50 random files in total. Then we make clients send file download requests randomly and let farmers report the number of steps needed for each VALUE_LOOKUP. We collect around 500 VALUE_LOOKUP records and compute mean of steps needed. The whole process is repeated for $n = 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150$ and $K = 1, 2, 3, 4, 5$.

From figure 5.2, we can observe that VALUE_LOOKUP also follows a logarithm curve, but the number of steps needed is far less than that needed by NODE_LOOKUP shown in 5.1. In a FileFarm network of 100 farmers, it only takes less than 3 steps to find a shard. Besides, as K increases, number of steps decreases roughly with a factor H_K depicted by Cai[20]. Putting this result with 5.2, we can conclude that a larger choice of K results in more uploading overhead while improving downloading performance, as NODE_LOOKUP and VALUE_LOOKUP are needed before uploading / downloading a shard, respectively.

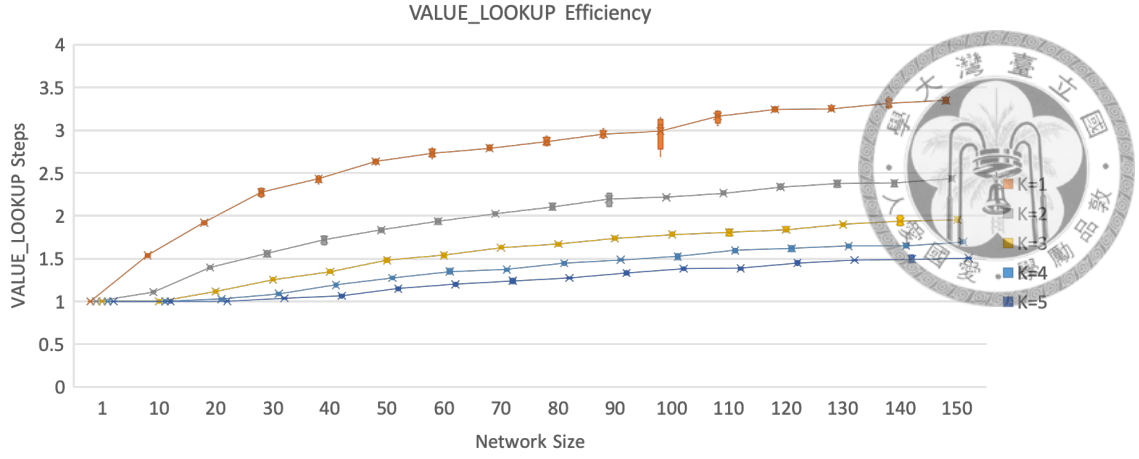


Figure 5.2: VALUE_LOOKUP steps with respect to redundancy parameter K and network size n . Like 5.1, VALUE_LOOKUP steps also grow with network size following a logarithm curve, while number of steps needed is far less than that needed by NODE_LOOKUP.

5.4 Experiment: Retrievability

Retrievability is one of the most important metrics used to measure quality of a cloud storage system. In this experiment, we want to examine how FileFarm is likely to hold client's data, and how certain configuration parameters affect the likelihood of uploaded files to be retrievable by clients. To be precise, we define retrievability as:

$$Retrievability = \frac{\#SuccessfullyReconstructed}{\#TotalDownloadAttempts}$$

Belows are the configuration parameter to be considered in this experiment:

1. α : Online probability of each farmer.
2. K : Number of redundant copies stored for each shard.
3. q : Redundancy parameter in a $(4, q)$ IDA schema; A file is recoverable given any 4 out of the $(4+q)$ shards.

As for the experimental procedure, we run 20 farmers with online probability α . After the farmers are bootstrapped, we run 500 clients and make them start random uploading/downloading. For each download request, the client reports if the file is successfully reconstructed. We collect 10,000,000 reports from clients and compute retrievability. The whole process is repeated for $K = 1, 2, 3$, $q = 0, 1, 2$ and $\alpha = 0.9, 0.99, 0.999, 0.9999$.

$\alpha = 0.9$	$K = 1$	$K = 2$	$K = 3$
$q = 0$	0.763826	0.980354	0.999486
$q = 1$	0.921600	0.995707	0.999987
$q = 2$	0.979964	0.998018	0.999970

$\alpha = 0.99$	$K = 1$	$K = 2$	$K = 3$
$q = 0$	0.9661670	0.9997400	0.9999470
$q = 1$	0.9944950	0.9999960	0.9999997
$q = 2$	0.9991700	0.9999980	0.9999994

$\alpha = 0.999$	$K = 1$	$K = 2$	$K = 3$
$q = 0$	0.9969620	0.9999982	0.9999978
$q = 1$	0.9996040	1	0.9999991
$q = 2$	0.9998940	1	1

$\alpha = 0.9999$	$K = 1$	$K = 2$	$K = 3$
$q = 0$	0.99977700	0.99999870	0.99999930
$q = 1$	0.99989955	0.99999990	0.99999920
$q = 2$	0.99997380	1	1

Table 5.1: Retrievability of files with respect to farmer's online probability α , network redundancy parameter K and IDA redundancy parameter q . The table shows that increasing K and q can both improve retrievability of files effectively.

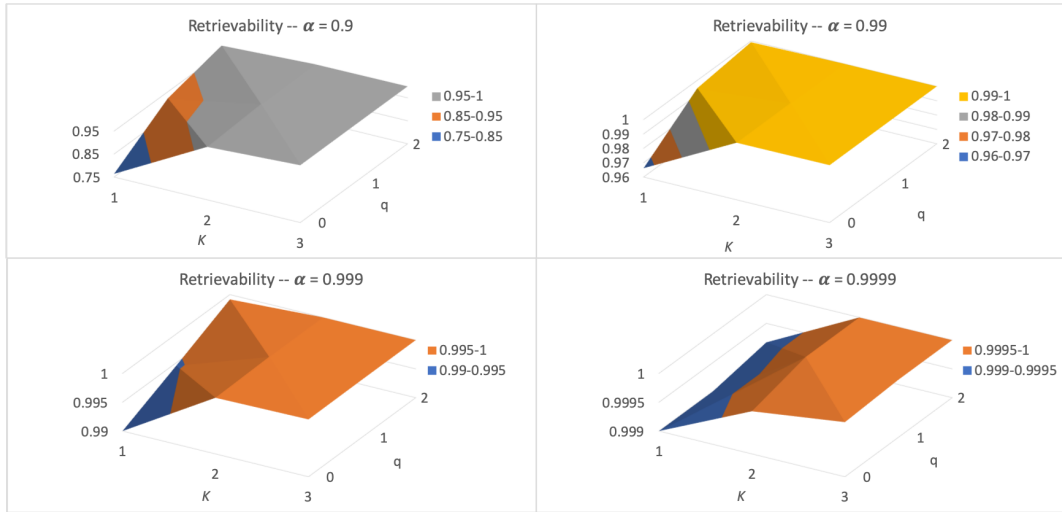


Figure 5.3: Retrievability of files with respect to α , K and q

From the result shown in 5.1, we observe that network redundancy parameter K and IDA redundancy parameter q both effectively impact the retrievability of files. By increasing K , retrievability can be improved significantly. However, increasing K has a relatively large overhead, considering the fact that an incremental in K will consume an extra amount of storage space equaling to the actual file size. To reduce cost while preserving retrievability, increasing q is a reasonable choice. In addition, we also observe that a selection of $K = 2$ roughly squares the data lost probability. For example, within a system where farmers have 0.1 probability to be offline (i.e. $\alpha = 0.9$), a setting of $K = 2$ and $q = 1$ roughly achieves the data lost rate of $0.1^2 = 0.01$, which is effective considering the fact that a public cloud service normally has much lower probability to be unavailable,

and we can make the probability of losing data even lower by building FileFarm upon them. Note that in some of the experimental cases where $\alpha \geq 0.999$, retrievability of files are estimated to be 1, which means we didn't observe any fail-to-download event in the 10,000,000 download attempts.



5.5 Experiment: Throughput

In this experiment, we want to test FileFarm's I/O performance under different file size and sharding schemas. As described in figure 4.3 and figure 4.4, the upload process of FileFarm involves slicing, encryption, IDA computation, NODE_LOOKUP, ... while the download process involves VALUE_LOOKUP, IDA computation, decryption, combining... Some of the operations can be done in parallel while others cannot. To analyze performance of such complicated flows, we run experiments, measure the elapsed time and compute throughput as follows:

$$Throughput = \frac{FileSize(MB)}{ElapsedTime(Sec)}$$

As for the experimental procedure, we run 5 farmers and 5 clients in total. Each farmer is assumed to have a 10 MB/s upload bandwidth. Once all farmers and clients are bootstrapped, we let clients start random upload and download files of size S with a sharding schema which generates p shards in total. While uploading and downloading, we let clients report time consumption for each operation until 100 upload and 100 download reports have been collected. We collect these reports and compute mean of upload/download throughput. The whole process is repeated for $S = 1, 4, 16, 64, 256, 1024$ and $p = 1, 2, 4, 8, 16, 32$.

From the result shown in figure 5.4, we observe that upload throughput grows when number of shards increases. This is due to the system design in which shards are uploaded in parallel. This growing trend achieves saturation when number of shards ≈ 16 , where incoming throughput of farmers achieves the NIC capacity limit of 10 Gbit/s. In the cases of smaller file size ($S = 1MB$ or $4MB$), the throughput curve goes down slightly with number of shards > 16 . This is caused by the preprocessing cost of NODE_LOOKUP,

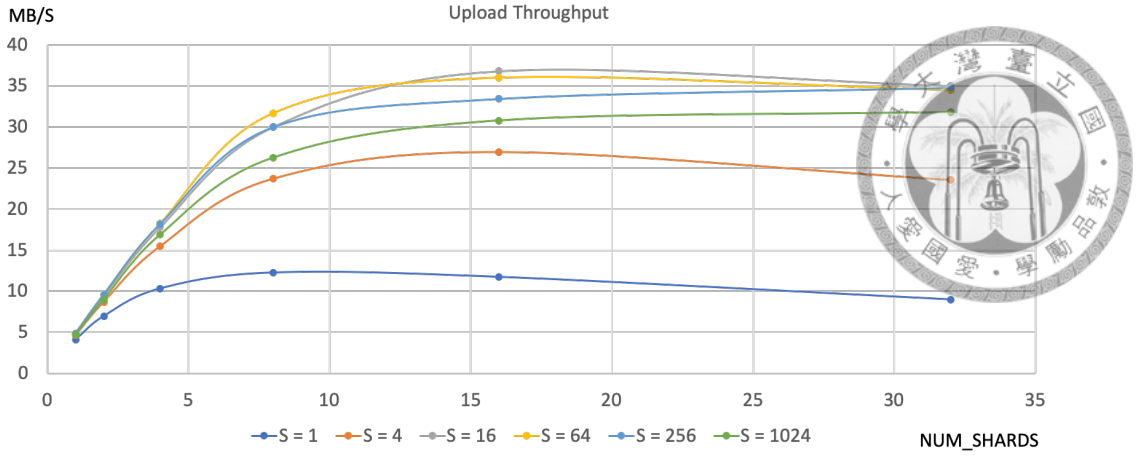


Figure 5.4: Upload throughput(MB/s) with respect to file size(MB) and number of shards p . Due to parallel upload design, upload throughput grows when number of shards increases.

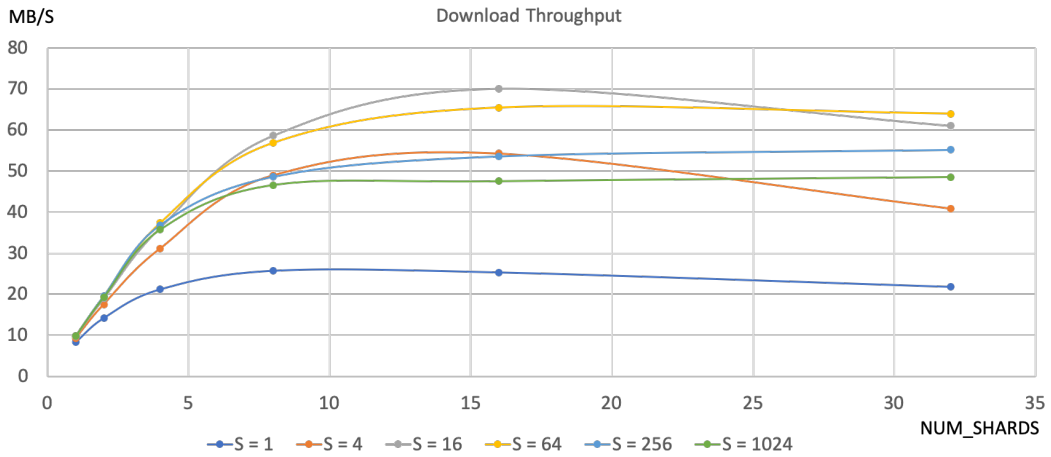


Figure 5.5: Download throughput(MB/s) with respect to file size(MB) and number of shards p . Download throughput roughly shares the same trends as upload throughput in terms of sharding schema and file size.

slicing, encryption, hashing, and IDA computation. When file size is relatively small, the preprocessing cost takes a significant term of timing overhead, while in cases of larger file size, the curve is mainly limited by farmers' NIC capacity.

Observing from chart 5.5, we discover that download throughput roughly shares the same trends as upload throughput in terms of sharding schema and file size. However, there is a noticeable difference: download throughput is much higher than upload throughput. This is because of the fact that download only involves retrieval of 1 copy of the shard instead of K ; thus, download operations induce less inter-farmer traffic than upload ones. In our experimental settings, number of physical hosts and NIC capacity on

each host are limited and are easily stuffed by frequent client requests. Thus, inter-farmer traffic tends to be bounded by hardware limitations and affect I/O performance. In real-world applications, however, the throughput bottleneck often occurs at client-side instead of farmer-side. Thus, upload/download throughput will mainly be determined by client bandwidth.



5.6 Experiment: Cost – Storage Release

The purpose of this experiment is to evaluate the cost-down effectiveness of *storage release* mechanism. By description in 4.7.1, *storage release* is a cost-reducing mechanism addressing redundant *static storage fee*, which comes up with scenarios of service outage or migration of storage providers. To investigate the effect of *storage release*, we simulate a migration scenario. In our scenario, the *static storage fee* is assumed to be 0.0197 US dollars per GB per month, which is the average value of 3 major public storage providers: Amazon S3, Microsoft S3, Google Cloud, as listed in table 1.1. The Kademlia redundancy parameter, K , is set to 2, which means for each shard, 2 redundant copies are stored in the network. The experimental procedure goes as follows: we first start a network of 3 public farmers and 50 clients. Clients are allowed to randomly upload files until a total of 2,000 files are stored in the network. With the file size set to 1GB and redundancy parameter K set to 2, the total stored data size is around 4 TB. After all of the files are stored in the network of 3 public farmers, we bootstrap another public farmer and let it join the network. Due to the *efficient key republishing* mechanism of Kademlia, some of the shards will start to be replicated to the newly-joined farmer. We then record the total *static storage fee* every simulational month until a year passed. We run this whole procedure in 3 different settings: (1) migration with *storage release*, (2) migration without *storage release*, (3) no migration, and plot the growing curve of total static fee charged by the public storage providers. Note that the non-migration case is a reference one in which 4TB of data are stored, with *static storage fee* being 0.0197 \$USD/GB/month and no additional farmer joining the network. In this reference case, the static fee is around 80 US dollars per month.

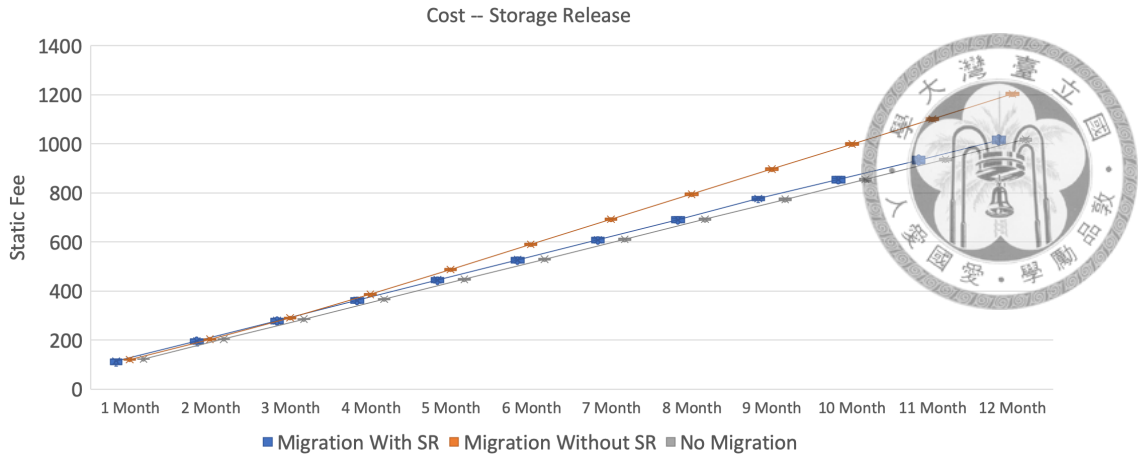


Figure 5.6: Growth curve of static fee in 3 different settings: (1) migration with *storage release*, (2) migration without *storage release*, (3) no migration. The non-migration case is a reference scenario in which 4TB of data are stored, with *static storage fee* being 0.0197 \$USD/GB/month and no additional farmer joining the network. The figure shows that when a new farmer joins, a 21.1% overhead of storage cost can be avoided by enabling *storage release*, which makes the cost curve sticking to that of the non-migration reference case.

From figure 5.6, we can observe that *static storage fee* grows linearly with time. This is due to our experimental settings that a fixed number of files are stored in the network, and thus monthly fees are charged according to the size of static files. Besides, we can also observe that in the case of *migration without storage release*, the cost trend is steeper than that of the other 2 cases, which implies that a static amount of redundant shards are stored on public clouds. These redundant shards then induce a fixed cost every month. In the context of this experiment where number of farmers migrates from 3 to 4, the storage overhead is roughly 21.1% more than the case where no migration occurs. By enabling *storage release*, this overhead can be avoided, with the cost curve sticking to that of the non-migration case.

5.7 Experiment: Cost – Prioritized Download

In this experiment, we want to evaluate the cost preserved by the *prioritized download* mechanism in a hybrid setting where both public clouds and private servers contribute to the storage network, while public clouds charge storage fees but private servers do not. As described in 4.7.2, the *prioritized download* mechanism saves *data transfer out fee* by pre-

ferring to download from private farmers rather than public ones. We conduct experiment according to such hybrid scenario: we first bootstrap 25 farmers, with N of them public farmers and other $(25 - N)$ of them private ones. Then we start 50 clients and let them upload and download files randomly, with the rate of 0.5 GiB download traffic per day. We record the transfer fee accumulated in the following 18 simulational months and compare the total *data transfer out fee* in 2 cases: (1) with *prioritized download*, (2) without *prioritized download*. The whole procedure is repeated for $N = 1, 2, 4$. In our scenario, the *data transfer out fee* is assumed to be 0.06 US dollars per GiB of download traffic, which is the average value of 3 major public storage providers: Amazon S3, Microsoft S3, Google Cloud, as listed in table 1.1.

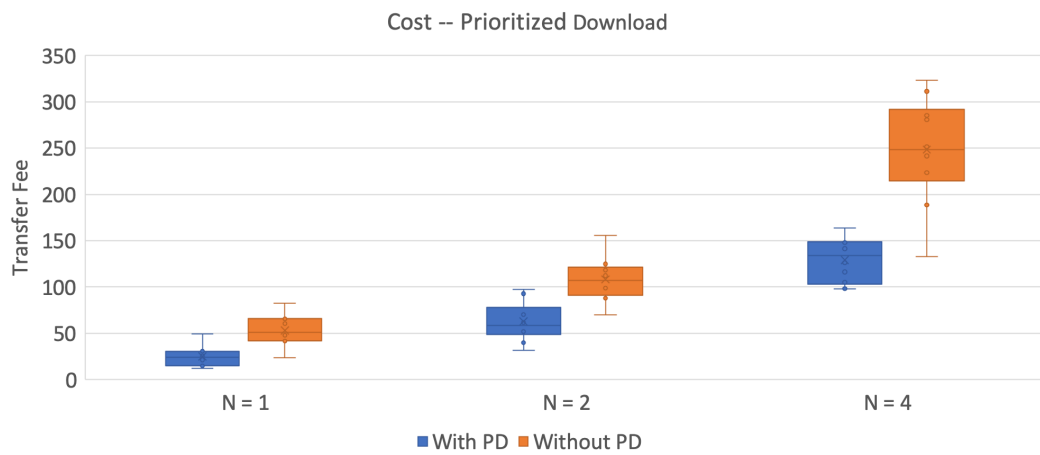


Figure 5.7: Comparison of transfer fee in 2 cases: (1) with *prioritized download*, (2) without *prioritized download*. In the scenario of *data transfer out fee* being 0.06 \$USD/GiB and client download traffic being 0.5 GiB/day, *prioritized download* mechanism preserves 47.6% of transfer fee in average.

From figure 5.7, we observe that *prioritized download* saves 52.3%, 42.0%, 48.0% of *data transfer out fee* in the cases of $N = 1, 2, 4$, respectively. Since *prioritized download* only changes the preference of downloading order, this saving of cost only applies to *data transfer out fee* but not *static fee*. With the same sense, *prioritized download* has no impact on retrievability of files since shards are still saved and available on the public clouds, only with lower access rates.



Chapter 6

Conclusion

This chapter makes a conclusion of the thesis. It starts by describing the problem FileFarm tries to solve. Then it summarizes the approach proposed by FileFarm. Finally it evaluates FileFarm from various aspects with experimental results.

6.1 Problem

Due to its high reliability and elasticity, cloud storage service has rapidly become the dominant adoption among all kinds of storage choices. However, for enterprises, saving data on a single cloud exposes their business to certain risks directly: (1) loss of control over data, (2) data leakage and insider attacks, (3) vendor lock-in problem. Thus, how to leverage existing cloud storage services without reliance on any single cloud has become an important research problem, which FileFarm tries to solve.

6.2 Approach

FileFarm solves the single-cloud dependence problem with a cloud-of-clouds architecture. To resolve the consistency and load-balancing issues caused by a centralized database design in conventional cloud-of-clouds work, FileFarm adopts a P2P strategy, in which each cloud, called farmer, synchronizes storage information with each other via Kademlia DHT protocol. Several desired properties of FileFarm are inherited from Kademlia: (1) redun-

dancy maintenance, (2) efficient search and (3) load-balancing design. However, in order to serve as an enterprise-level storage, 4 further properties are required but cannot be provided by Kademlia: (1) data confidentiality, (2) access management, (3) cost-efficiency, (4) retrievability. FileFarm meets these requirements by designing corresponding mechanisms: (1) *encryption and Information Dispersal Algorithm*, (2) *decentralized authentication*, (3) *storage release and prioritized download*, (4) *public farmer ID assignment*. These mechanisms collectively make FileFarm a robust, secure and cost-efficient storage solution.

6.3 Evaluation

Properties of FileFarm are tested through experiments in a wide variety of aspects. Experiments 5.2 and 5.3 show that lookup steps in FileFarm grow with network size following a logarithm curve. With the number of farmers growing from 50 to 100, it only takes around 2 more steps to locate all K closest farmers, which ensures routing efficiency and scalability of FileFarm. Experiment 5.4 shows that FileFarm effectively improves retrievability of data by redundancy designs of 2 parameters: (1) K : Number of redundant copies stored for each shard (2) q : redundancy parameter in a $(4, q)$ Information Dispersal Algorithm schema. Within a system where farmers have 0.01 probability to be offline, a setting of $K = 2$ and $q = 1$ achieves the data lost rate of $0.01^2 = 0.0001$. Experiment 5.5 shows that FileFarm effectively improves throughput by parallel operations, with a peak value of 37 MB/s uploading speed and 70 MB/s downloading speed under the constraints of 10 MB/s farmer bandwidth and 1 GBits/s NIC capacity. Experiment 5.6 shows that FileFarm's *storage release* mechanism effectively reduces the redundant cost of *static storage fee*. In a migration scenario where number of public farmers increases from 3 to 4, a 21.1% overhead of storage cost can be avoided by enabling *storage release*, which makes the cost curve sticking to that of the non-migration reference case. Experiment 5.7 shows that FileFarm's *prioritized download* mechanism preserves *data transfer out fee* effectively. In a hybrid scenario of 4 public farmers and 21 private farmers, 48% of transfer fee can be preserved by *prioritized download*.

Through these experiments, we evaluate the claimed properties of FileFarm in terms of lookup efficiency, scalability, retrievability, throughput and cost efficiency. These properties ensure that data in FileFarm are distributed safely over multiple clouds with optimized cost. As long as there are no concurrent failures occurring at more than $K-1$ farmers, the data will still be retrievable. Besides, once a farmer churns off, the data it used to store will be replicated to other farmers automatically, which maintains consistent level of redundancy. In case of a new farmer joining the network, storage load on each farmer will be balanced automatically. The properties collectively make FileFarm a robust, secure and cost-efficient storage solution. In addition to system design, we also implement a proof-of-concept that not only confirms our claims but also serves as a product prototype of our structured P2P file storage solution.






Bibliography

- [1] The state of software-defined storage, hyperconverged and cloud storage. https://s3.amazonaws.com/ydtimages/~yourdai7/wp-content/uploads/2017/08/18133654/SDS_2016_Report_FINAL.pdf, 2017. [Online; accessed 27-June-2019].
- [2] Amazon Simple Storage Service. <https://aws.amazon.com/s3/>, 2019. [Online; accessed 27-June-2019].
- [3] aMule. <http://www.amule.org>, 2019. [Online; accessed 02-July-2019].
- [4] Apache ZooKeeper. <https://zookeeper.apache.org>, 2019. [Online; accessed 10-July-2019].
- [5] BitTorrent. <https://www.bittorrent.com/bittorrent-free>, 2019. [Online; accessed 02-July-2019].
- [6] eMule. <https://www.emule-project.net/home/perl/general.cgi?l=1>, 2019. [Online; accessed 02-July-2019].
- [7] Ethereum. <https://www.ethereum.org>, 2019. [Online; accessed 02-July-2019].
- [8] Google Cloud Storage. <https://cloud.google.com/storage/>, 2019. [Online; accessed 27-June-2019].
- [9] IPFS. <https://ipfs.io>, 2019. [Online; accessed 02-July-2019].
- [10] Microsoft Azure Cloud Storage. <https://azure.microsoft.com/en-us/product-categories/storage/>, 2019. [Online; accessed 27-June-2019].

- [11] Napster. <https://us.napster.com>, 2019. [Online; accessed 02-July-2019].
- [12] RFC 4158. <https://tools.ietf.org/html/rfc4158>, 2019. [Online; accessed 02-July-2019].
- [13] Storij. <https://storj.io>, 2019. [Online; accessed 02-July-2019].
- [14] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240. ACM, 2010.
- [15] M. A. AlZain, E. Pardede, B. Soh, and J. A. Thom. Cloud computing security: From single to multi-clouds. In *2012 45th Hawaii International Conference on System Sciences*, pages 5490–5499, Jan 2012.
- [16] M. A. AlZain, B. Soh, and E. Pardede. MCDB: using multi-clouds to ensure security in cloud computing. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 784–791. IEEE, 2011.
- [17] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.
- [18] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.
- [19] C. Cachin, R. Haas, and M. Vukolic. Dependable storage in the intercloud. *IBM research*, 3783:1–6, 2010.
- [20] X. S. Cai and L. Devroye. A probabilistic analysis of Kademlia networks. In *International Symposium on Algorithms and Computation*, pages 711–721. Springer, 2013.
- [21] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.



- 
- [22] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, page 21, 2012.
- [23] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
- [24] M. Li, C. Qin, and P. P. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 111–124, 2015.
- [25] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- [26] P. Mell, T. Grance, et al. The NIST definition of cloud computing. 2011.
- [27] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.
- [28] K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, Jan 2012.
- [29] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.
- [30] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

- [32] C. Taylor. Survey reveals tech trends reshaping data storage.
<https://www.enterprisestorageforum.com/storage-management/survey-reveals-tech-trends-reshaping-data-storage.html>, 2018. [Online; accessed 27-June-2019].
- [33] M. Vukolic. The Byzantine empire in the intercloud. *SIGACT News*, 41(3):105–111, 2010.
- [34] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.

