EFFECTIVE CLOUD STORAGE IN DISTRIBUTED SYSTEM USING KEY-VALUE STORES

G.BHAVANA[1] & P.BALAKISHAN[2] & Dr.M.SUJATHA[3]
[1]M.Tech Student, [2]Associate Professor, [3]Associate Professor
Department Of CSE, JYOTHISHMATHI INSTITUTE OF TECHNOLOGICAL SCIENCES, KARIMNAGAR T.S.INDIA.

Received: May 22, 2018 Accepted: July 08, 2018

ABSTRACT *Nowadays, cloud-based storage services are rapidly growing and becoming an emerging trend in data storage field. There are many problems when designing an efficient storage engine for cloud-based systems with some requirements such as big-file processing, lightweight meta-data, low latency, parallel I/O, deduplication, distributed, high scalability. Key-value stores played an important role and showed many advantages when solving those problems. This work reports on the realization of a key manager that uses an untrusted distributed key-value store (KVS) and offers consistent key distribution over the Key-Management Interoperability Protocol (KMIP). To achieve confidentiality, it uses a key hierarchy where every key except a root key itself is encrypted by the respective parent key. The hierarchy also allows for key rotation and, ultimately, for secure deletion of data. The design permits key rotation to proceed concurrently with key-serving operations. A prototype was integrated with IBM Spectrum Scale, a highly scalable cluster file system, where it serves keys for file encryption. Linear scalability was achieved even under load from concurrent key updates. The implementation shows that the approach is viable, works as intended, and suitable for high throughput key serving in cloud platforms.*

Keywords: *Cloud Storage, Key-Value, NoSQL, Big File, Distributed Storage.*

I. INTRODUCTION

Cloud-based storage services commonly serves millions of users with storage capacity for each user can reach to several gigabytes to terabytes of data. People use cloud storage for the daily demands, for example backing-up data, sharing file to their friends via social networks such as Facebook [3], Zing Me [2]. Users also probably upload data from many different types of devices such as computer, mobile phone or tablet. After that, they can download or share them to others. System load in a cloud storage is usually really heavy. Thus, to guarantee a good quality of service for users, the system has to face many difficult problems and requirements: Serving intensity data service for a large number of users without bottle-neck; Storing, retrieving and managing big-files in the system efficiently; Parallel and resumable uploading and downloading; Data deduplication to reduce the waste of storage space caused by storing the same static data from different users. In traditional file systems, there are many challenges for service builder when managing a huge number of bigfile: How to scale system for the incredible growth of data; How to distribute data in a large number of nodes; How to replicate data for load-balancing and fault-tolerance; How to cache frequently accessed data for fast I/O, etc. A common method for solving these problems which is used in many Distributed File Systems and Cloud Storages is splitting big file to multiple smaller chunks, storing them on disks or distributed nodes and then managing them using a meta-data system [1], [8], [24], [4]. Storing chunks and meta-data efficiently and designing a lightweight meta-data are significant problems that cloud storage providers have to face. After a long time of investigating, we realized that current cloud storage services have a complex meta-data system, at least the size of metadata is linear to the file size for every file. Therefore, the space complexity of these meta-data system is O(n) and it is not well scalable for big-file. In this research, we propose new big-file cloud storage architecture and a better solution to reduce the space complexity of meta-data.

Key-Value stores have many advantages for storing data in data-intensity services. They often outperform traditional relational databases in the ability of heavy load and largescale systems. In recent years, key-value stores have an unprecedented growth in both academic and industrial field. They have low-latency response time and good scalability with small and medium key-value pair size. Current key-value stores are not designed for directly storing big-values, or big file in our case. We executed several experiments in which we put whole file-data to key-value store, the system did not have good performance as usual for many reasons: firstly, the latency of put/get operation for big-values is high, thus it affects other concurrent operations of key-value store service and multiple parallel accesses to different value reach limited. Secondly, when the value is big, there is no more space to cache another objects in main memory for fast access operations. Finally, it is difficult to scale-out system when number of users and data increase. This research is implemented to solve those problems when storing big-values or big-file using key-value stores. It brings many advantages of key-value store in data management to design a cloud-storage system called Big File Cloud (BFC).

These are our contributions in this research: –

- Propose a light-weight meta-data design for big file. Every file has nearly the same size of meta-data. BFC has O(1) space complexity of meta-data of a file, while size of meta-data of a file in Dropbox[1], HDFS[4] has space complexity of O(n) where n is size of original file. See Fig 9
- Propose a logical contiguous chunk-id of chunk collection of files. That makes it easier to distribute data and scale-out the storage system.
- Bring the advantages of key-value store into big-file data store which is not default supported for big-value. ZDB is used for supporting sequential write, small memory-index overhead.

These contributions are implemented and evaluated in Big File Cloud (BFC) that serve storage for Zing Me Users. Disk Image files of VNG's CSM Boot diskless system are stored in Big File Cloud.

Encryption plays a fundamental role for realizing secure networked computing environments. Key management ensures reliable and secure distribution of cryptographic keys to legitimate clients, which are then able to encrypt data or to establish secure communication channels. Key management for cloud scale distributed installations poses additional challenges over classical, centralized systems, due to the vastly bigger systems and the higher demands for resilience and security. Maintaining the confidentiality of encryption keys is extremely important, especially for encrypting data in storage systems, where losing access to the encryption key implies losing the data itself. A communication system, in contrast, may just restart the session if a key is lost. As key management is critical for many environments, industry standards have been introduced to separate key management functions from the components that consume keys, and to consolidate key lifecycle management at centralized, well-protected systems [1]. Key management can be seen as an essential service of an IT infrastructure and especially for cloud platforms, similar to network connectivity, computing, and storage. The most prominent standard for distributed key management today is the OASIS Key Management Interoperability Protocol (KMIP) [2], which specifies operations for managing, storing, and retrieving keys at a remote server. For local key management using library-style access PKCS #11 [3] is the prevalent interface. In the context of cloud services, where service interactions are REST calls, the open-source Barbican [4] key manager provides keys to all services of OpenStack. Commercial cloud platforms use proprietary protocols inside their infrastructure. Key managers differ according to the operations they support and in terms of their performance, resilience, and security. Prominent commercial key servers often put emphasis on the needs of enterprise environments, such as fine-grained authentication and support for hardware security modules (HSMs). For example, governmental standards for handling health data dictate a reliable audit trail to reconstruct all operations accessing cryptographic keys. Enterprise key managers are also designed for high availability to allow uninterrupted service. They must support the complete lifecycle of cryptographic secrets, with operations for creating, importing, storing, reading, updating, exporting, and deleting keys. Designing and operating a key-management service in a distributed system with many entities running cryptographic operations is challenging because it must balance between the conflicting goals of performance and security.

In this paper we present a solution for scaling a key management service to cloud applications with thousands of clients and possibly millions of keys. This includes the capability to scale dynamically while maintaining security. Our distributed key management solution should handle all core key lifecycle-management tasks and scale in a linear way. Existing enterprise-grade key managers do not provide such scalability because they rely too heavily on centralized components, such as HSMs or strongly consistent relational databases. In particular, we address key management for an enterprise environment with a scalable cloud platform. The clients or endpoints accessing the key manager primarily perform dataat-rest encryption, but could be any other application that executes cryptographic operations. The key server has to provide a consistent view of the available keys when accessed from multiple clients.

The key manager consists of three components with different security assumptions:

Trusted storage: A master key resides on a trusted medium for persistent storage across power cycles. When the system is turned off, no component apart from the trusted storage medium maintains any relevant cryptographic information; in other words, only the root of trust must be guarded. For supporting lifecycle management and key rotation, the medium should support secure erasure. Potential implementations are hardware security modules (HSMs), USB-attached smart cards, or also cheap removable USB drives that can be physically destroyed for key erasure. For better scalability, multiple instances of trusted storage may be utilized.

Metadata key-value store: For supporting scalable operation, all cryptographic material apart from the master key is held in an untrusted distributed storage system modeled as a key-value store (KVS). All data stored in the KVS is protected with the master key, i.e., by wrapping or through a key hierarchy. The distributed KVS supports high-throughput and scalable access to the stored key material. The KVS platforms available today often do not support atomic operations but only eventual consistency, however. This can lead to issues caused by simultaneous access to the same key by multiple clients.

Key server: Keys are delivered to the clients in cleartext and are available in the memory of the key server. The key server is trusted not to disclose keys to unauthorized clients. It overwrites keys no longer needed according to standard practice.

The KVS used for scalability does not permit strongly consistent operations on keys, such as key rotation. We resolve this through a novel design for concurrent key-management operations that use a weakly consistent data store. In particular, we describe an implementation of a key manager using hierarchically protected keys, where only the master key resides on trusted storage. The important features of this design are:

1) The key server permits continuous access to keys in the presence of ongoing key rotation operations;

2) The key server scales linearly with the available server resources. Experiments show that the design and prototype deliver the expected performance, in particular, linear scalability with the server resources.

B. Products and related work Every cryptographic system must manage keys. Several standalone key managers have been developed to provide a generic service and support the standard protocols, such as KMIP [2]. Key-lifecycle management operations are important for enterprise deployment [1] and for satisfying regulatory requirements [5]. Two prominent products addressing this space are the Vormetric Data Security Management (DSM) server [6] and the IBM Security Key Lifecycle Manager (ISKLM). Vormetric DSM offers interoperability with KMIP and facilitates integration with database systems, such as Microsoft SQL Server. It is certified at several FIPS 140-2 levels, depending on the hardware on which it operates. With an appropriate HSM, certification ranges up to FIPS 140-2 Level 3. Due to its centralized architecture it is inherently not scalable. ISKLM runs on standard servers atop a software stack that includes IBM Web Sphere and DB2. For key storage, ISKLM may use a PKCS #11-attached HSM, and for resilience and high availability, it can run distributed on a cluster. It provides a web interface for management, control, and auditing, and a KMIP interface serving keys to endpoints. Due to its dependence on the database, ISKLM is also centralized. Although these and other similar solutions are complex and incur high operational expenses, they do not scale as well for distributed platforms. Barbican [4] inside Open Stack is a scalable key management component for cloud platforms. It supports asynchronous operations on symmetric and private keys, public keys, and certificates. As all services in Open Stack, it is accessed through a REST API and uses the Keystone [7] component for authorization. Its operations are not as general as those of KMIP, for instance. The backend of Barbican is modular, designed as a plug-in system, so that it currently supports an SQL database or a KMIP client for storing secrets remotely, or an HSM accessed through PKCS #11 for local key storage. In Barbican, multiple workers run concurrently and on different hosts, but they all interact with the central backend database through a message queue. This limits its scalability. Considering also its restricted functionality in the API, Barbican doesn't offer the desired scalable support for key storage in combination with key rotation and secure erasure. Several key management services are available in cloud platforms today, such as Amazon KMS, CloudHSM, and IBM Key Protect [8], [9], [10]; they hide the complexities of inhouse key management but rely on partial trust in the cloud operator.

II. RELATED WORKS

LevelDB [10] is an open source key-value store developed by Google Fellows Jeffrey Dean and Sanjay Ghemawat, originated from BigTable [5]. LevelDB implements LSM-tree and consists of two MemTable and set of SSTables on disk in multiple levels. When a key-value pair is written, it firstly is appended to commit log file, then it is inserted into a sorted structure called MemTable. When MemTable's size reaches its limit capacity, it will become a read-only Immutable MemTable. Then a new MemTable is created to handle new updates. Immutable MemTable is converted to a level-0 SSTable on disk by a background thread. SSTables which reach the level's limit size, will be merged to create a higher level SSTable. We already evaluated LevelDB in our 6 prior work [16] and the results show that LevelDB is very fast for small key-value pairs and data set. When data growing time-by-time and with large key-value pairs, LevelDB become slow for both writing and reading.

Zing-database (ZDB) [16] is a high performance keyvalue store that is optimized for auto increment Integer-key. It has a shared-memory flat index for fast looking-up position of key-value entries in data files. ZDB supports sequential writes, random read. ZDB is served in ZDBService using thrift protocol and distribute data using consistent-hash method. In BFC, both file-id and chunk-id are auto increment integer keys, so it is very good to use ZDB to store data. The advantage of ZDB is lightweight memory index and performance for big data. When data grow it still has a low-latency for read and write operation. Many other researches try to optimize famous data structures such as B+tree [14] on SSD, HDD or hybrid storage device. It is also useful for building key-value stores on these data structures. With the design and architecture of BFC, the chunkId of a file has a contiguous integer range, ZDB is still the most effective to store chunk data. Distributed Storage Systems (DSS) are storage systems designed to operate on network environment including Local Area Network(LAN) and the Internet. In DSS, data is distributed to many servers with ability to serve millions of users. DSS can be used to provide backup and retrieve data functions. BFC fully supports these functions. DSS also provide services as a general purpose file system such as NFS or other Distributed File System (DFS) such as GFS [11]. BFC is a persistent non-volatile cloud storage, so it can provide this function in Linux by using FUSE and BFC client protocol. Applications store data on BFC can take advantages of its high performance and parallel processing ability.

III. OBJECTIVES

A. Security goals

In a deployment of the key manager, there are two actors: the key manager and the client. Only the key manager may access the master key and the trusted storage. Any other entity is subsumed into an adversary, an actor with complete access to the metadata KVS, communication links, and so on. The adversary might be an untrusted storage provider, the client itself, or the operator of the network between the client and the server. Clients can authenticate themselves to the key manager and establish a secure connection using TLS, with server- and client-side certificates. The key manager authorizes access to keys based on the identity in the client certificate and delivers keys to the client over the secure channel. There are two concrete goals addressing security.

Key confidentiality: The adversary should not be able to learn any useful information about a key to which it does not have access, i.e., keys for which it is not authorized, neither by interacting with the server, by observing the KVS, nor by listening on the network.

Key erasure: Clients can request that a key is deleted permanently. This supports the secure deletion of data protected by this key, since physical deletion of stored data from disks or solid-state storage is no longer possible in practice [11]. Key deletion supports the approach to secure deletion introduced by Di Crescenzo et al. [12] and recently extended by Cachin et al. [13], where only a single key at the root of a hierarchy must actually be erasable. All other metadata in the hierarchy can be exposed to the adversary. From the perspective of the key manager, secure deletion is supported through key rotation of the master key and deletion of expired keys.

B. Other goals

Scalability: The service performance should scale linearly with the available resources and not disrupt client operations. To achieve this, the key manager is implemented by parallel stateless workers that have access to the master key. The key server processes have to operate in a consistent way, to serve the current version of a key consistent with the data protected by it. It should also be possible to dynamically adapt the number of servers to the load of the system.

Availability: Associated with scalability, the service must also be resilient to failures of individual nodes. Problems with the consistency of different key management servers can result in loss of data, when different clients encrypt with different keys.

Usability: In any security system, the weakest link is the erroneous handling of sensitive material by users or operators (as the reliance on user-memorable passwords shows). The system should support operations in a straightforward and transparent way.

IV. DESIGN

A. Architecture

The architecture of the key server is shown in Figure 1. Every node essentially runs the same key server independently of the others, but has access to the distributed metadata KVS and to the trusted storage. Having multiple key servers work in parallel provides scalability. The core server locally caches the most recently accessed keys in memory. The KVS for metadata must support the usual put and get operations on key/value pairs, with the additional requirement that it supports conditional put. More precisely, the KVS supports

get(key) → (value, version) put(key, value, version) → success

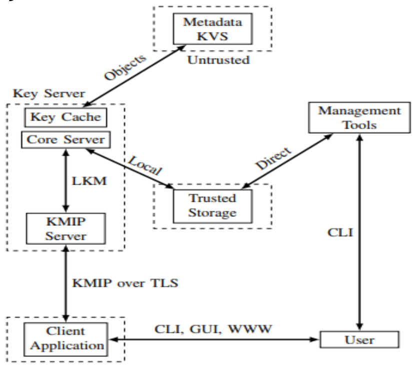

Fig. 1. Overview of the key manager components. Cryptographic material is only stored an handled inside the dashed regions. where put takes effect only when success = TRUE. Every put operation monotonically increments the version associated to a value. The specification requires that put returns TRUE if and only if the version of key that is replaced by the operation was version. The prototype that targets file encryption in IBM Spectrum Scale uses the cluster configuration repository (CCR) as the distributed KVS implementation, which is a Spectrum-Scale-internal distributed data store with high availability that offers versioned put/get operations.

Clients address keys through a unique, randomly generated identifier in the KMIP interface. More user-friendly attributes such as a name may be used to associate a key with the function that the key plays in the context of the application, and KMIP permits multiple keys to have the same name in the scope of a KMIP server. The name, along with other attributes, are part of the metadata of the key itself, and these attributes can be used to retrieve the identifier of a key, which is then used for subsequent operations. The communication between the client and the key server is performed using the tag-type length-value (TTLV) transport of the KMIP protocol, which in turn is communicated over a mutually-authenticated TLS connection.

Cryptographic keys stored in the KVS are protected using a master key retrieved from trusted storage. More precisely, every key in the KVS is wrapped with the master key using AES and authenticated encryption with Galois Counter Mode (GCM). For every wrapped key (in the KVS), also the identifier of the wrapping key is stored in the metadata. This is needed for key rotation. The trusted storage accessed by every node is initialized with the master key. It can be evolved through key rotation. A user interacts with the system through management tools for the provisioning of the master key in trusted storage. The core of the key server provides a local key manager (LKM) interface towards a KMIP proxy server, which interfaces to the clients. This ensures compatibility with many existing endpoint clients in cloud and enterprise storage systems that access a key manager over KMIP. The key management service itself is completely stateless, and can flush its cache at any time.

A client accesses a key over KMIP (via Create, Locate, Get, Destroy, Rekey . . . operations), using the identifier, which is stored e.g., in the metadata of the data unit protected by this key. For a cloud object store, this could be in the metadata of the account/container/object of an object as in OpenStack Swift; for a distributed file system, this could be in the inode of a file. Storing keys in wrapped form ensures the key confidentiality requirement. No data on persistent storage, apart from the trusted storage, reveals any information about the keys. Decrypted keys only exist in volatile memory. Furthermore using authenticated encryption for wrapping also guards against accidental or intentional alteration of cryptographic keys or their attributes in the KVS. The goals of scalability and availability are achieved by the design of stateless key server nodes, which can be placed into different failure zones, and by the use of a highly available KVS for storing the key data. The inherent scalability and availability of the distributed KVS is further enhanced by deploying separate trusted storage on each key server node, thus improving the characteristics of the entire system. A potential drawback of this design is the overhead related to the management of the trusted storage. The added complexity comes from the fact that the master keys are now managed internally by the system, and not delegated to an external key management service. The overhead varies depending on e.g., the choice of trusted storage media type, the degree of integration of the trusted storage management with the rest of the system, and regulatory compliance requirements.

B. Operations

Cryptographic material is stored in a format that is a simplification of the network-level representation of KMIP. This has the advantage that much of the needed functionality is already available from existing KMIP libraries. Clients authenticate to the KMIP server during the TLS handshake using a client certificate, following the standard practice that has been available and deployed with many KMIP servers. After authentication, they may exchange data in the KMIP format over the secure channel.

At system initialization time, the core server generates a fresh master key and its identifier, and writes them to the trusted storage over the local interface. Every key is stored in the KVS as a separate data object under its identifier. During operation the core server responds to client operations, arriving through the KMIP interface, as follows. Create: The key material is generated from a cryptographically strong random source. The key itself is wrapped with the master key; the master-key identifier together with the key identifier are added as attributes. The data is serialized in the KMIP-derived format and stored as an object in the KVS, according to its key identifier. Key and key identifier are returned to the client. Get: For retrieving keys, the server uses the identifier provided in the request by the client to locate the key. To obtain the key material from the KVS, the server retrieves the object containing the wrapped key, unwraps the key material with the master key specified in the attributes, and verifies the integrity of the unwrapped data. Then it returns key and key identifier to the client. Referencing a key with the unique identifier instead of the name supports the implementation of client-side key rotation; note that the name may be the same across different versions. The key server additionally maintains a cache with the recently served keys in cleartext. The server only has to query the KVS when it has no cached copy of the key or when a cached key reaches its refresh age. Destroy: The server deletes the corresponding object with the wrapped key in the KVS. Note this does not yet securely erase the key according to the security model, as the KVS may still keep a copy of the object and the master key has not yet changed. This is addressed below.

Rekey: The KMIP Rekey operation for a given key creates a fresh key with the same name and other attributes as the existing key. It is implemented by creating a new key and destroying the old one, and by removing the name from the old key. The returned key material is fresh and has a new identifier. The old key may still be accessible until the master key is changed as well. Secure deletion: The key server provides secure deletion in the sense that the master key is rotated. Any key material that was stored on the untrusted KVS wrapped with the previous master key then becomes inaccessible. To support this, master keys contain a version in their attributes. The challenge lies in rotating the master key without disrupting the other operations of the key server. For the moment, assume that the rotation is triggered by a dedicated agent, which may be a special management node or an administrator client. For rotating one particular key in a hierarchy, we call the keys that it wraps children (i.e., a clientvisible key) and the wrapping key the parent (i.e., master key).

To rotate the parent (i.e., master) key, fresh key material with a higher version is first chosen and written to trusted storage. From this moment on, any operation on the parent key that does not specify a version or identifier returns the new key. On the other hand, for any children wrapped with the old key, the appropriate version can be retrieved with the key identifier available in the metadata of the wrapping. This ensures continuous operation while rotation is in progress, in particular, key creation, retrieval, and destruction operations can be served by other key-manager nodes. The agent now cycles over all children of the parent, unwraps the child key, rewraps it with the new parent key, and stores the outcome in the corresponding object. When this is complete, the old parent key is securely erased from the trusted storage. This design can be generalized to key-wrapping hierarchies of depth larger than one in a future extension. When the key-rotation agent invokes a rotation operation for a key, the rotation recursively trickles down to all its children. Key rotation would then progress over the tree of children, until they are all wrapped with the new key.

The only limitation regarding concurrent operation is that the agent performing key rotation must be unique, as otherwise, two rotation operations might occur concurrently and leave the data in the KVS in an inconsistent state. With an arbitrary, weakly consistent KVS, one can implement this by partitioning the key space across the agents or alternatively by using an external locking mechanism. With a versioned KVS, however, the KVS-level objects can be replaced conditionally on the old version containing the previous wrapping key identifier. This ensures that each key is rotated atomically, with the same parent key as for all of its siblings. The operations for rotating the master key ensure the security goal of key erasure.

V. CONCLUSION

The data deduplication method of BFC uses SHA-2 hash function and a key-value store to fast detect data-duplication on server-side. It is useful to save storage space and network bandwidth when many users upload the same static data. In the future, we will continue to research and improve our ideas for storing big data structure in larger domain of applications, especially in the ”Internet of things” trend.

As encryption of data at rest becomes more prevalent, the challenge of managing the encryption keys also surfaces for diverse systems. The scalable key-management design presented in this work targets cloud-scale deployments. It is compatible with storing the master keys in an HSM, but achieves better performance than a solution exclusively relying on a centralized key manager or a HSM. The key manager is built on top of an untrusted key-value store (KVS) and demonstrated in the context of the IBM Spectrum Scale cluster file system. It serves file-encryption keys using the KMIP standard. A key-hierarchy and key rotation operations supporting secure deletion of critical data have been described and prototyped. The evaluation shows that the key manager was able to scale linearly even under load from key updates, and performance measurements conducted on the individual components indicate that the throughput and latency are mostly limited by the performance of the distributed KVS.

REFERENCES

1. M. Bjorkqvist, C. Cachin, R. Haas, X. Hu, A. Kurmus, R. Pawlitzek, and ¨ M. Vukolic, “Design and implementation of a key-lifecycle management ´ system,” in Proc. Financial Cryptography and Data Security (FC 2010), ser. Lecture Notes in Computer Science, R. Sion, Ed., vol. 6052. Springer, 2010, pp. 160–174.
2. OASIS Key Management Interoperability Protocol Technical Committee, “Key Management Interoperability Protocol Version 1.2,” 2015, oASIS Standard, available from http://www.oasis-open.org/committees/ documents.php?wg abbrev=kmip.
3. OASIS PKCS 11 Technical Committee, “PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40,” 2016, oASIS Standard, available from https://www.oasis-open.org/committees/ tc home.php?wg abbrev=pkcs11.
4. “OpenStack Barbican,” https://wiki.openstack.org/wiki/Barbican, 2018.
5. E. Barker, “Recommendation for key management — Part 1: General,” National Institute of Standards and Technology (NIST), NIST Special Publication 800-57 Part 1 Revision 4, 2016, available from http://csrc. nist.gov/publications/PubsSPs.html.
6. “Vormetric Data Security Management,” https://www.thalesesecurity. com/products/data-encryption/vormetric-data-security-manager, 2018.
7. I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking personal cloud storage. In Proceedings of the 2013 conference on Internet measurement conference, pages 205–212. ACM, 2013.
8. I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In Proceedings of the 2012 ACM conference on Internet measurement conference, pages 481–494. ACM, 2012.
9. P. FIPS. 197: the official aes standard. Figure2: Working scheme with four LFSRs and their IV generation LFSR1 LFSR, 2, 2001.
10. S. Ghemawat and J. Dean. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb. Accessed November 2, 2014.
11. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In ACM SIGOPS Operating Systems Review, volume 37, pages 29–43. ACM, 2003.
12. Y. Gu and R. L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. Computer Networks, 51(7):1777–1799, 2007.
13. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In Proceedings of the 2010 USENIX conference on USENIX annual technical conference, volume 8, pages 11–11, 2010.
14. P. Jin, P. Yang, and L. Yue. Optimizing b+-tree for hybrid storage systems. Distributed and Parallel Databases, pages 1–27, 2014.
15. D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. Computer Networks, 31(11):1203–1213, 1999.