Distributed Enctypted File System

Jiarui Gao[1]
[1]School of Computer Science, Shanghai Key Laboratory of Intelligent Information Processing,
Fudan University, China
jrgao14@fudan.edu.cn

ABSTRACT

In this paper, I propose a Distributed Encrypted File System (DEFS) which allows clients to store large scale data on a cluster of untrusted servers. All files in DEFS are stored in a distributed and encrypted manner. It provides exclusive account for each client, and offers full recoveries for damaged files.

KEYWORDS

Distributed File System; Distributed Data Storage; Encrypted FIle System.

ACM Reference format:
Jiarui Gao[1]. 2017. Distributed Enctypted File System. In *Proceedings of , , ,* 5 pages.
DOI: http://dx.doi.org/xxx

1 INTRODUCTION AND RELATED WORK

Distributed Encrypted File System (DEFS) is designed to combine encryption and distributed storage. Through DEFS, clients can store large scale data on untrusted servers. DEFS provide encryption services to prevent servers from stealing any clients' information. Also, it offers full recoveries if any clients' file is damaged by malicious servers. The distributed storage gives DEFS the capacity of hosting large scale of data. And the lock server in the system keeps a high consistency for DEFS.

1.1 Encrypted File System

Many non-networked file systems aims at keeping clients' data secure at a system level. An early example for Unix is Cryptographic File System (CFS)[2]. A modern classic design of encrypted file system is SUNDR[5], which is a network file system designed to store data securely on untrusted servers. The design in DEFS is largely inspired by this work.

1.2 Distributed File System

Typical traditional distributed file systems are NFS[6] and AFS[4]. Main difference between them is that AFS obtains better performance by caching the files, and it sacrifices the synchronization. NFS is later optimized to be Serverless Network File Systems[1]. Hadoop Distributed File System[7] and Google File System[3] concentrates on the support of large scale data.

,
© 2017 ACM. ACM ISBN xxx...$15.00
DOI: http://dx.doi.org/xxx

2 SYSTEM OVERVIEW

This section introduces the overview design of Distributed Encrypted File System (DEFS). As illustrated in Fig. 1, DEFS is composed of two parts, namely Distributed File System (Sec. 4) and Encrypted Layer (Sec. 3).

Encrypted Layer guarantees that all the data (including file content ,filename and client's directory structure) in DEFS is encrypted. All file content is signed by the client, what insures that any modification will be detected. Before any execution, it examines authority for every command carefully, which keeps clients' accounts secure from illegal operations from possible malicious users. Also, all illegal attempts will be logged.

Distributed File System consists of LockServer, one NameNode and a cluster of many DataNodes. LockServer provides read-write lock to maintain consistency. The NameNode/DataNode design is largely inspired by Hadoop [7] and Google File System[3]. DEFS stores all metadata on one trusted server (NameNode), and stores application data blocks on others untrusted servers (DataNodes). NameNode records mappings with regards to file blocks and DataNodes, and provide block-location related services. DataNodes stores the assigned block replicas.

3 ENCRYPTED LAYER DESIGN

As shown in Fig. 2, the Encrypted Layer provides all encryption-related services. It is transparent to both clients and servers, and it insures all the information stored on servers are encrypted. The login manner offers separate accounts for each client, which prevents clients from possible malicious users as well. All illegal attempts made by clients will be recorded in a special log file by Encrypted Layer. The following subsections introduces detailed designs in this layer.

3.1 Account Initialization

Each newly registered client is offered one pair of RSA key (n_1, e_1, d_1) and one AES key K. The client needs to keep the private key (n_1, d_1) by himself. And the public key (n_1, e_1) is stored in the system. The the system uses (n_1, e_1) to encrypt the AES key to be K_{en}, and stores it in the system. To be specific, the system maintains a mapping between clients' names and their corresponding keys' locations. RSA public key (n_1, e_1) is later used for client authentication and encrypting file full paths (filename included). AES key K is taken to encrypt file contents.

When a client want to login to DEFS, he must provide both the client name and the location of his private key (n_1, d_1). If it matches with (n_1, e_1), then the authentication is granted. The purpose of using RSA private key directly to login and persevering only the encrypted version of backup file is to insure that there is no sensitive information in the Encrypted Layer.

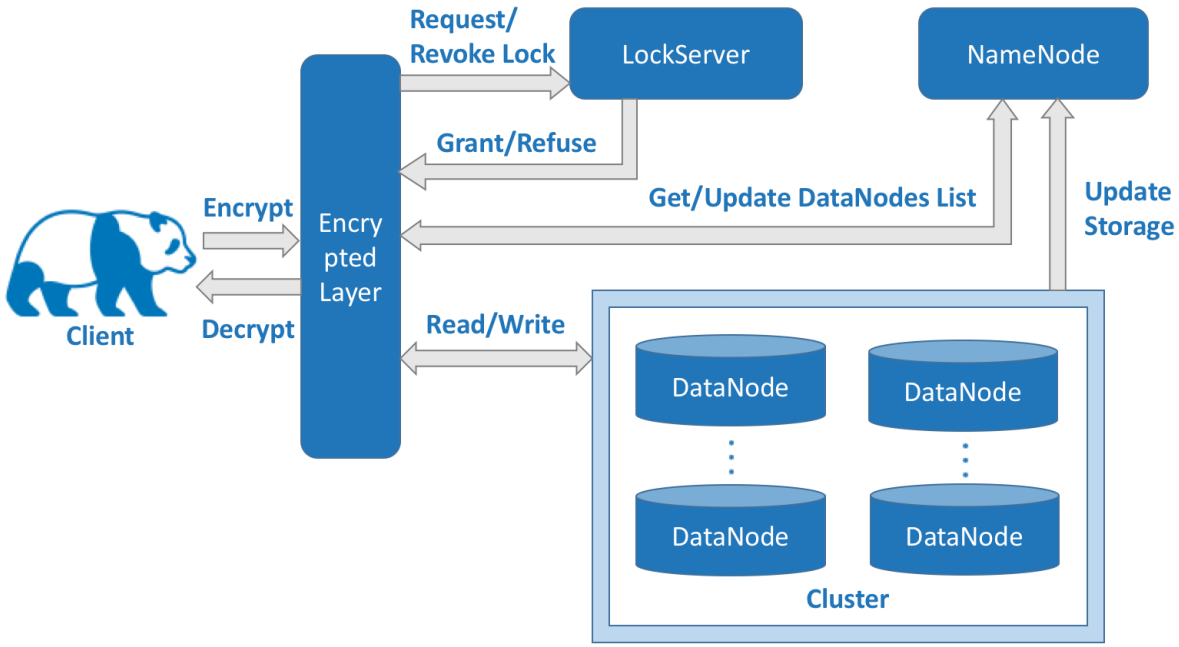

Figure 1: An overview of Distributed Encrypted File System (DEFS). The system contains an Encrypted Layer (left part) and a distributed file system (right part). The Encrypted Layer provides encryption-related services for clients. Generally, all operations in distributed system requires getting corresponding locks from LockServer first, and the locks are revoked after each operation. After granted a lock, the client ask NameNode for IP addresses where the file blocks are stored (a list of DataNodes). Then the client write/read data blocks from DataNodes directly. To ensure that NameNode offers reasonable allocations of DataNodes, each DataNode sends Update Storage message to NameNode after any modification immediately.

3.2 File Uploading/Writing

When a client requests to Upload/Write a file, the Encrypted Layer makes sure all the information is encrypted according to following procedures.

- File full paths with filenames included(to insure uniqueness of each file) are encrypted with (n_1, e_1) as their new names on the server.
- The system uses client's provided private key (n_1, d_1) to decrypt (K_{en}) to get AES key K. File content is encrypted with K as new content stored on the server.
- Client's private key (n_1, d_1) is used to sign the hash value of the whole file content. Then the signature is added at the beginning of the encrypted file. By now, the encryption of file is finished.
- A backup of the encrypted file is stored in the file system for its future recovery.
- Send Upload/Write request to the Distributed File System.

3.3 File Downloading/Reading

When a client requests to Download/Read a file, the Encrypted Layer decrypted the received file from Distributed File System in following steps.

- First, get the encrypted file name. The system uses (n_1, e_1) to encrypt file's full path (filename included) to get the encrypted filename stored on the server.
- Then check the signature. The system decrypted the signature stored at the beginning of the file with public key (n_1, e_1). And if the result equals to the hash value of the encrypted file content, the authentication is granted. Otherwise, the systems take the illegally modified by untrusted servers into consideration, and after notification to the client, the backup version in the system will be uploaded automatically to the Distributed File System.
- The system uses private key (n_1, d_1) to decrypt (K_{en}) and get AES key K. The it uses K to decrypt file content.
- The decrypted file is downloaded is ready to be read.

3.4 Other Commands

The system provide other commands including *Move, Copy, Remove, etc.* Those commands could be converted to Upload/Write (Sec. 3.2) and Download/Read (Sec. 3.3). For example, command *Move* includes

- Download the file.
- Move (Rename) the backup version.
- Upload the new file.

3.5 Layer Content

As introduced in former in Sec. 3, there is some client-related data stored in directories in the Encrypted Layer. DEFS insures that even the Encrypted Layer is invaded entirely by hackers somehow, nothing will be leaked.

All data stored in Encrypted Layer includes:

- Mapping of (client name, location of corresponding RSA public key).
- Mapping of (client name, encrypted AES key).
- Log file recording all illegal attempts.
- Clients' RSA public keys.
- Clients' backup file with encrypted filename and encrypted file content.

As shown above, the Encrypted Layer is totally secure.

4 DISTRIBUTED FILE SYSTEM DESIGN

Fig. 3 demonstrate the design of Distributed File System. It mainly consists of three parts, namely LockServer, NameNode, and a cluster of DataNodes. The system design provide a distributed storage for large scale data.

4.1 LockServer

Generally, I consider the situation in DEFS as a typical readers-writers problem[8]. The LockServer provide service following readers-preference solution. That is to say once the first reader is in the entry section, it will lock the resource. Doing this will prevent any writers from accessing it. Subsequent readers can just utilize the locked (from writers) resource. The very last reader (indicated by the readcount variable) must unlock the resource, thus making it available to writers.

When requesting for read or write locks, the client send a control message ($encrypted filename, locktype$) to LockServer, and in return he gets the granted lock or a waiting message (1/0). After the operation, the client will revoke the lock via another control message to the server ($encrypted filename, locktype$).

4.2 NameNode

The single NameNode in Distributed File System maintains all the metadata and is in charge of allocating data blocks to specific DataNodes. As illustrated in Fig. 3, in the DEFS design, there are two mappings. DEFS keeps the entire mapping both in RAM and in json files to improve durability.

A DEFS client wanting to download/read a file first contacts the NameNode for the locations of data blocks (DataNode IDs) comprising the file and then reads block contents directly from the DataNode closest to the client. When a client requests to upload/write a file, a message asking for file allocation or file location update is sent to NameNode. Then the NameNode nominate three DataNodes for each block and return their ID list back to the client.

To guarantee the selected DataNodes have the capacity to store the block replicas, NameNode updates the storage mapping immediately upon receipt of control messages from DataNodes. During the nomination, NameNode randomly selects three DataNodes for each block among which have enough storage.

4.3 DataNodes

Distributed File System contains a cluster of DataNodes to store data blocks. In DEFS, all DataNodes are considered not trustworthy, and the information sent to them are only encrypted data blocks. The data pipeline is between clients and DataNodes. DataNodes process the request from clients directly.

During the startup process of each DataNode, it performs a handshake with NameNode to inform the connection and provides its current storage capacity. And after data blocks are written or deleted, DataNodes send control message ($DataNodeID, Storage$) to NameNode to update the storage left.

4.4 Fault Tolerance

DEFS provides a relevant robust storage on untrusted servers. As long as the NameNode is trustworthy, the DataNodes can be untrusted. That is to say the block replicas hosted by DataNodes may be modified or deleted at anytime. DEFS has two methods in dealing with such situation.

Random Selection of DataNodes. As mentioned in Sec. 4.2, the NameNode nominate DataNodes randomly for each block. Assume there are n DataNodes totally, and m of them is malicious. Here malicious DataNodes modify any data block sent to it. We also assume that the client's file contains k blocks, and the number of replicas in DEFS is r. Then the possibility of unrecoverable damage to one data block is $\frac{C_m^r}{C_n^r}$. The possibility of unrecoverable damage to the whole file is:

$$1 - (1 - \frac{C_m^r}{C_n^r})^k \tag{1}$$

As designed in DEFS, $r = 3$. And if $\frac{m}{n} = \frac{1}{3}$, which means one third of the DataNodes are malicious, the possibility of recovering the full file still remains to be:

$$(1 - \frac{C_m^r}{C_n^r})^k > (1 - (\frac{m}{n}))^k > (\frac{26}{27})^k \tag{2}$$

As calculated, DEFS keeps a high tolerance of fault.

File Backup. As introduced in Sec. 3.2, the Encrypted Layer maintains a backup version for the clients, which is absolutely practical when the file size is small. In the decryption procedure, if the retrieved file is found to be modified, the backup version will be used to correct the data on DataNodes immediately. This backup method provides a 100% percent recovery of any file.

5 IMPLEMENTATION

All the mentioned design of DEFS is fully implemented in Python, and library *Crypto* is used for encryption. Source code is available at https://github.com/kittenish/Distributed-Encrypted-File-System.

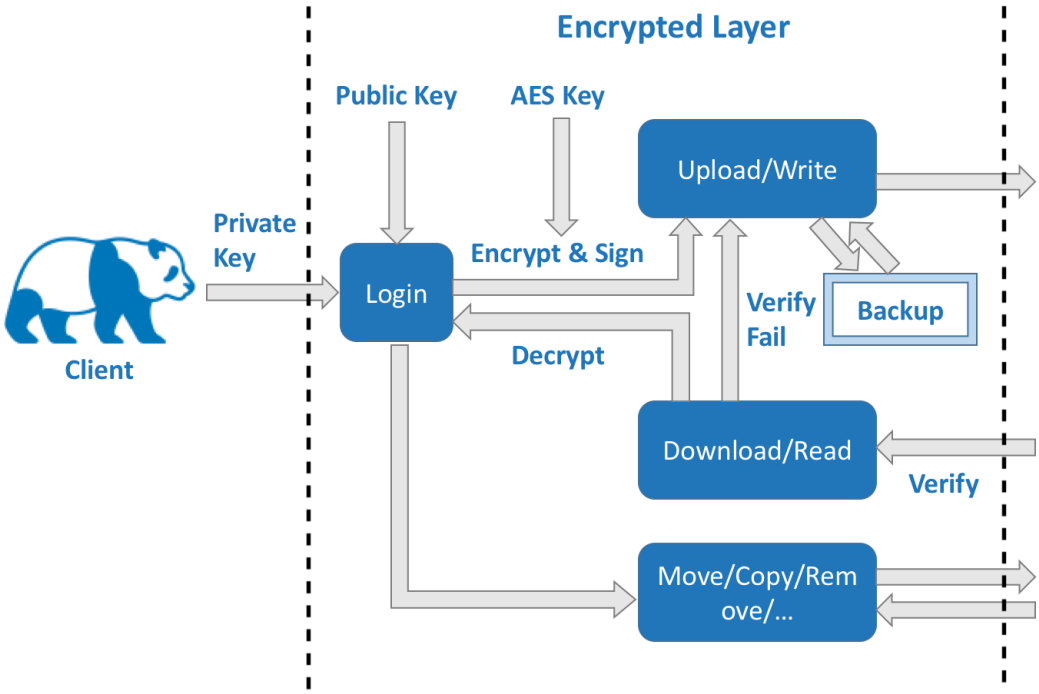

Figure 2: An overview of the Encrypted Layer. The client needs to provide the private key to login to the system. The public key and the encrypted AES key is stores by the system. A simple work flow is shown in the figure.

Screen shots for command line interfaces are presented in Appendix Sec. A. This section will introduce detailed parameter settings and side information about DEFS.

5.1 Parameter Settings

The number of DataNodes is set to be 5, and their initial storage is $100MB$. The number of replicas is 3. All the information communication is through socket. Besides user name and the private key, each client needs to provide a available socket port to login. The file backup method is adopted for fault tolerance. And all metedata are stored as *json* format in local file system. ALso, DEFS tries to catch exceptions as much as possible to avoid crush.

5.2 Side Information

Ideally, socket connection between clients and DataNodes are supposed to be established when the transportation of data steam begins. But for simplicity, in my implementation, all socket connections between clients and other servers are built upon *login* command. The *login* request will fail if the client's socket port is unavailable or any of the initial connections fail.

Both the Encrypted Layer and the distributed storage is transparent to clients. Commands of reading and writing is the same as traditional file systems. The command interface is designed to be user-friendly, and clear help information is available for every function.

6 FUTURE WORK

Lots of extension could be made in DEFS to achieve better performance. In this section, both design improvement and implementation modification will be discussed.

From design aspect:

- Clients' separate accounts prevent files from being shared, the sharing function should be added.
- When file damage is detected, the client should report to NameNode. And more functions could be added to NameNode to detect malicious servers and disconnect with them when necessary.
- More clusters of DataNodes can be added to improve storage capacity.
- The LockServer currently uses a readers-preference solution, which may cause severe starvation for writers. Other lock granting and revoking algorithms can be adopted.

From implementation aspect:

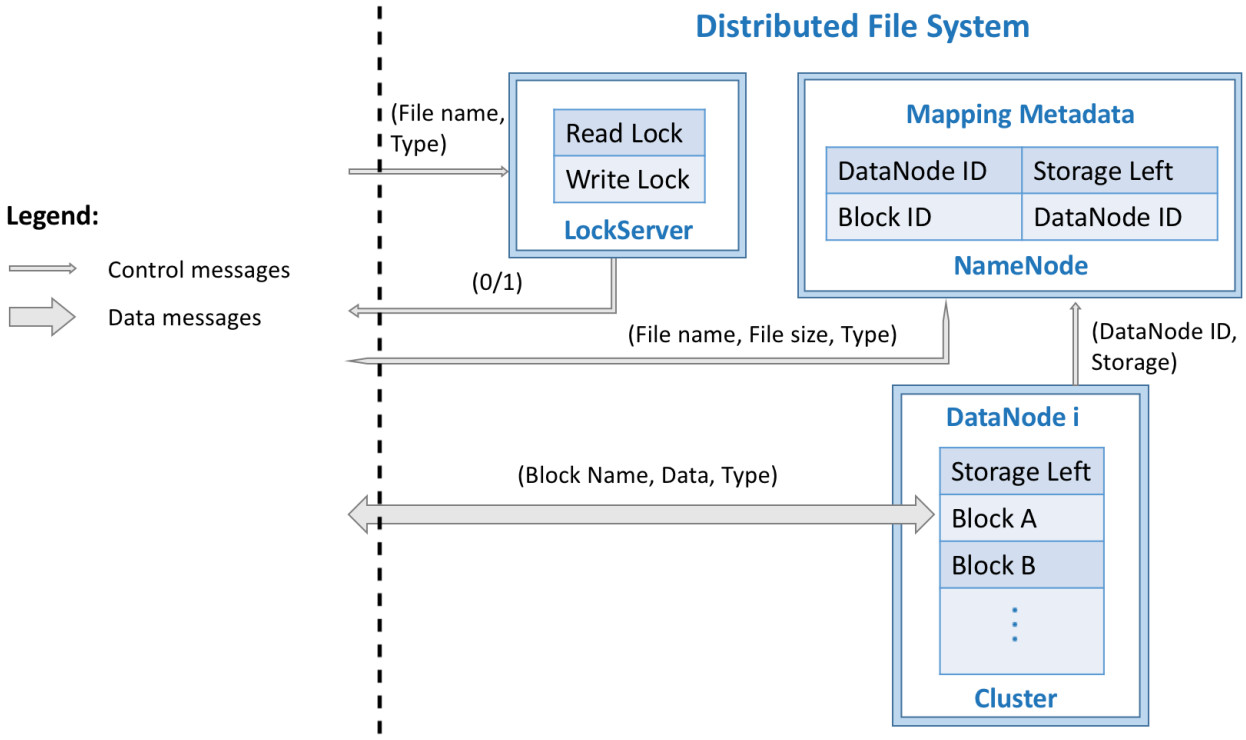

Figure 3: An overview of Distributed File System (DFS). The LockServer provides read/write locks. The NameNode stores the mapping metadata shown in the table. And all the data pipelines are directly between clients and DataNodes.

- Reduce the hard coded definitions in the system, and make DEFS configurable.
- More rules in Encrypted Layer for examining commands' authority to avoid malicious users.
- When file damage happens, Encrypted Layer should try to recover the data from blocks inside DEFS first to get the latest version, before uploading the backup version.

7 CONCLUSION

In conclusion, DEFS is a file system designed to store large scale data on untrusted servers. Clients' data is hosted in a distributed and encrypted manner. And all the information is protected from both servers and other malicious users.

A APPENDIX

Here are some screen shots in DEFS.

REFERENCES

[1] Thomas E Anderson, Michael D Dahlin, Jeanna M Neefe, David A Patterson, Drew S Roselli, and Randolph Y Wang. 1995. Serverless network file systems. In *ACM SIGOPS Operating Systems Review*, Vol. 29. ACM, 109–126.
[2] Matt Blaze. 1993. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM conference on Computer and communications security*. ACM, 9–16.
[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 29–43.
[4] John H Howard and others. 1988. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center.
[5] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. 2004. Secure Untrusted Data Repository (SUNDR).. In *OSDI*, Vol. 4. 9–9.
[6] Spencer Shepler, Mike Eisler, David Robinson, Brent Callaghan, Robert Thurlow, David Noveck, and Carl Beame. 2003. Network file system (NFS) version 4 protocol. *Network* (2003).
[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 1–10.
[8] Wikipedia. 2017. Readers writers problem. (2017). https://en.wikipedia.org/wiki/Readers-writers_problem