Managing Data Retention Policies at Scale

Jun Li, Sharad Singhal, Ram Swaminathan, Alan H. Karp

HP Laboratories
HPL-2010-203

Keyword(s):

large-scale policy management; compliance and regulatory; data retention; encryption key store; cloud service

Abstract:

Compliance with regulatory policies on data remains a key hurdle to cloud computing. Policies such as EU privacy, HIPAA, and PCI-DSS place requirements on data availability, integrity, migration, retention, and access, among many others. This paper proposes a policy management service that offers scalable management of data retention policies attached to data objects stored in a cloud environment. The management service includes a highly available and secure encryption key store to manage the encryption keys of data objects. By deleting the encryption key at a specified retention time associated with the data object, we effectively delete the data object and its copies stored in online and offline environments. To achieve scalability, our service uses Hadoop MapReduce to perform parallel management tasks, such as data encryption and decryption, key distribution and retention policy enforcement. A prototype deployed in a 16-machine Linux cluster currently supports 56 MB/sec for encryption, 76 MB/sec for decryption, 31, 000 retention policies/sec read and 15,000 retention policies/sec write.

External Posting Date: December 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: December 21, 2010 [Fulltext]
To be published in IFIP/IEEE International Symposium on Integrated Network Management 2011, Dublin, Ireland, May 23-27, 2011.

Managing Data Retention Policies at Scale

Jun Li, Sharad Singhal, Ram Swaminathan, and Alan H. Karp
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304, USA
{jun.li, sharad.singhal, ram.swaminathan, alan.karp}@hp.com

***Abstract*— Compliance with regulatory policies on data remains a key hurdle to cloud computing. Policies such as EU privacy, HIPAA, and PCI-DSS place requirements on data availability, integrity, migration, retention, and access, among many others. This paper proposes a policy management service that offers scalable management of data retention policies attached to data objects stored in a cloud environment. The management service includes a highly available and secure encryption key store to manage the encryption keys of data objects. By deleting the encryption key at a specified retention time associated with the data object, we effectively delete the data object and its copies stored in online and offline environments. To achieve scalability, our service uses Hadoop MapReduce to perform parallel management tasks, such as data encryption and decryption, key distribution and retention policy enforcement. A prototype deployed in a 16-machine Linux cluster currently supports 56 MB/sec for encryption, 76 MB/sec for decryption, 31,000 retention policies/sec read and 15,000 retention policies/sec write.**

Keywords-large-scale policy management; compliance and regulatory; data retention; encryption key store; cloud service

I. INTRODUCTION

Cloud computing is a promising paradigm to offer IT cost reductions and business agility improvements. However, compliance with regulatory policies still remains a key hurdle to wide adoption of cloud computing [27]. The service environment has to manage data owned by the customers according to mutually agreed data management policies in order to ensure compliance with regulatory policies such as the Data Protection Directive under EU privacy law [11], HIPAA [16], and PCI-DSS [21]. These regulatory policies are often translated to enforceable or auditable actions, such as data availability, data integrity, data migration, data retention, and data access.

Although these compliance and regulatory requirements are not new and have been addressed in traditional enterprise computing environments, addressing them in the context of cloud services introduces new challenges, one of which is scalable management and enforcement of policies. Imagine a backup service that offers data retention to 300 enterprises, with each enterprise having 10^4 users, and each user owning 10^5 files. Such a service must be capable of managing $3x10^{11}$ files. If each file is encrypted with a 32-byte key, the key store itself will require over 10 TB. A service capable of managing medical records for the entire U.S. population of 300 million people, with each person owning 10^3 records requires the same scale.

Our objective is to build a scalable policy management service with the ultimate goal of managing 10^{11} data objects. This paper details the design and prototype implementation of a policy management service that is primarily focused on data retention. Data retention belongs to a class of data policies called *action policies* that specify what to do under the current situation [18]. We believe that other policies such as data backup, data archiving, and data migration [4, 5], can be expressed in action policies and enforced similarly.

Managing data policies at scale poses various challenges, including the following two that we believe are key challenges to managing data retention policies:

- **Scalable Policy Enforcement**: A scalable engine that supports policy enforcement in real time for data access, and updates at the rate data changes is required. State information [24] and contextual information associated with data objects need to be tracked for policy enforcement. Enforcement needs to be carried out reliably as machine failure in large systems is common.
- **High Availability and Security of Management Metadata**: Policy management related metadata, which can include data management artifacts like encryption keys, audit logs [2], state and context information for policy decisions, should be treated as being as critical as the data being managed. High security and high availability of such management related metadata are actually required by compliance regulatory policies such as HIPAA and PCI-DSS.

Additional challenges include, for example, how to manage scalable relationships defined between data policies, between data objects, and between data policies and data objects. Runtime correlation and decision making require complex data/policy relationships be captured thoroughly, structured efficiently, and evaluated quickly. Yet another challenge can be multi-tenancy. The volume of customers introduces a new dimension of complexity and scalability. A common policy management service needs to manage customer specific and data specific policies. Multi-tenancy introduces issues related to concurrent data access, combinatorial relationship explosion (e.g., cross-organizational access rights delegation [24]), customer-specific data policies, and customer data compartmentalization [7]. Because data retention policies are rather simple in relationship expression and policy evaluation, this paper will not focus on how these additional challenges are addressed in managing data retention policies.

In our data retention management service, each file can be associated with a retention policy. An inter-data-center secure and reliable encryption key store holds encryption keys for each file under retention management. The encryption key controls the lifetime of the file. By removing the encryption key, all the file copies in online and offline environments become unrecoverable. Our policy store, encryption key store, and other data management metadata stores, are implemented with a scalable structured data store [6, 10, 15]. To achieve scalable management, the data management tasks, which include encryption and decryption, key distribution and policy enforcement, are performed concurrently in a machine cluster with the MapReduce framework [9, 14]. The prototyped cloud service has been deployed on a 16-machine Linux cluster with 128 cores (8 cores per machine), which supports 56 MB/sec for encryption, 76 MB/sec for decryption, 31,000 retention policies/sec read and 15,000 retention policies/sec write.

With respect to the two key challenges identified earlier, our data retention management service demonstrates that:

- Policy management at scale requires metadata management at scale. We have developed a highly available and highly secure cross-data-center encryption key store as a part of the policy management service to specifically manage encryption keys, a particular type of metadata essential to control data lifetime and data access.
- During policy enforcement and management task execution, failures can happen in a machine cluster and leave behind side-effects to data objects and policy enforcement states. Side-effects cannot be handled automatically by built-in recovery capabilities of Hadoop MapReduce. We developed a state-aware retry execution scheme to implement the Map and Reduce functions of each policy management task. This recovery scheme allows execution to continue in the next round from the recorded persistent states and does not rely on transactional support that can significantly reduce overall system scalability.

The rest of the paper is structured as follows. Section 2 introduces the data retention management system. Section 3 shows the architecture that exposes the management system as a service, and how Hadoop and MapReduce are used to perform management tasks at scale. Section 4 details our design of the highly secure and available encryption key store. Section 5 shows how various failure scenarios are handled. Section 6 reports our service prototype's running environment and performance measurements. We contrast our policy management service with related systems in Section 7 and conclude the paper in Section 8.

II. DATA RETENTION MANAGEMENT

Corporate data retention policies demand that enterprise data should remain accessible up to a certain time, and afterwards be deleted permanently with no recoverable trace. Timely removal not only allows the enterprise to manage sensitive data in compliance with regulatory policies, but also reduces storage costs of ever-growing data [7].

Many solutions exist for record retention [5, 17, 20, 4], but none has been demonstrated at the scale we envision. Furthermore, two key concerns have not been addressed in existing solutions. First, current solutions frequently ignore off-site data on removable media such as tapes. Tracking and managing such off-site information assets is challenging, and often becomes the root cause of data breaches to sensitive data. Secondly, ensuring deletion of data becomes hard once data is replicated to multiple storage tiers or sites to achieve high availability [4, 5]. Often there is no central point of control that can guarantee deletion of all copies at the proper time.

Figure 1 shows a file-based data retention management system that addresses the above two concerns. We encrypt data at rest, and use the encryption key to control the lifetime of the file object, as introduced in the revocable backup system [3]. By centrally managing encryption keys, the service can effectively manage both on-line and off-line files. Once an encryption key is destroyed, all on-line and off-line copies become instantaneously unrecoverable. This mechanism both protects data against breaches, and provides an effective way of making off-site data unusable.

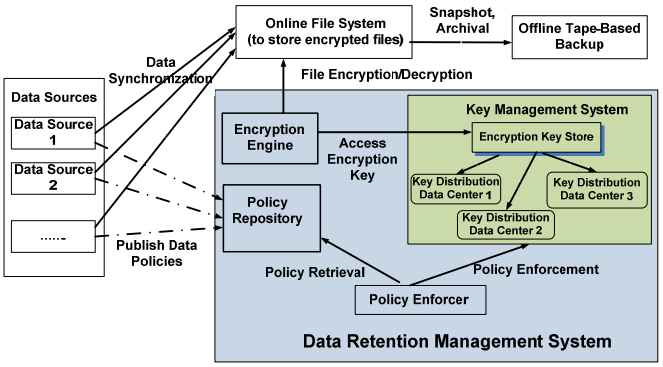

Figure 1. Data retention policy management for file-based data.

Data sources synchronize their data with the *Online File System*. Each file is encrypted by the *Encryption Engine* with a unique symmetric key for the life of the file. The encrypted files are stored online for fast retrieval. They can be further archived to offline media if their access becomes infrequent, but due to retention policies, they still need to be preserved for a long time (e.g., medical images often must be kept for more than ten years). The *Key Management System* provides a highly secure and available key store to hold the encryption keys. The key store itself is never backed up to offline media to ensure that keys that are destroyed are unrecoverable. The *Policy Enforcer* periodically scans the *Policy Repository* to determine keys with expired retention times, and deletes them from the key store. Other enforcement actions can include the removal of the encrypted file from online media to reclaim the storage, and notification of the deletion action to the file owner. Since all files are encrypted, the Online File System can be outsourced to a less secure environment such as Amazon's Simple Storage Service (S3) [1].

A scalable storage service such as S3 [1] has file related access control policies stored as part of the file metadata and enforced at data access points (e.g., service front-end). Data retention is enforced when retention time expires and a designated policy enforcer has to be responsible for such event detection and policy enforcement, instead of relying on data access points that are only activated to respond to data access requests. The scalable encryption engine and the scalable encryption key store shown in Figure 1 provide special functionalities for data retention, but in general are not required by a scalable storage system which is focused more on performance, reliability and availability [1, 4, 5].

A data retention policy can be specified in one of two formats: an absolute future time instant when the retention time expires, or as the time for data retention after the last data access or update. Figure 2 shows an example of expressing the data retention policy as an obligation policy in Ponder [8]. In this example, the retention policy is applied to a domain which represents all the files being managed. Due to transient failures, multiple retries may be required and the maximum number of retries (5, for example), needs to be set.

```
type oblig+ retention (target t) {
   do t.deleteKey()->t.reclaimSpace()->t.notify();

   on Time.before (
         t.policy().expirationTime(), Time.now());
   when t.retryEnforcementTimes() < 5;
}

inst oblig+ retentionPolicy1= retention (/files);
```

Figure 2. Data retention policy expressed in Ponder.

III. Service Architecture for Data Retention

We developed our retention service using Hadoop, an open-source software platform to support reliable, scalable and distributed processing. Hadoop has a reliable and distributed file system called HDFS that follows Google File System [13]. Hadoop supports MapReduce [9] to perform scalable data processing on a machine cluster. Users define data processing logic in the Map and Reduce functions and the input data and output data are both stored in HDFS. HBase [15] is a scalable structured data store that follows Google's BigTable [6]. Other scalable file or block based stores [1] and structured data stores [6, 10] can provide functionality and scalability similar to HDFS and HBase. Our service architecture chose HDFS and HBase mostly due to better integration among HDFS, MapReduce and HBase. In particular, HBase uses HDFS as its persistent store and MapReduce can directly accept HBase tables as input data sources.

Figure 3 shows the overall architecture of our policy management service. The service architecture is horizontally scalable. We use HDFS for the persistent file store and HBase for the backend structured data store to store data retention policies, encryption keys and status tracking information. We use MapReduce for the scalable policy management engine. The backend service architecture is exposed as a Web Service with four access APIs: (1) file access (upload, download, update); (2) policy access (create, update, read); (3) encryption

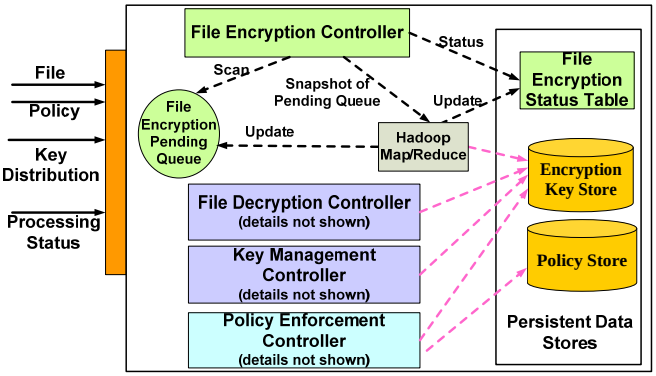

Figure 3. Data-retention service implemented with Hadoop and HBase.

key management across data centers including key distribution and key reconstruction; and (4) status queries on management tasks such as file upload and download, key distribution and reconstruction for long running, batch and asynchronous processing tasks. Correspondingly, the processing engine consists of four task controllers for (1) file upload and encryption, (2) file decryption and download, (3) retention policy enforcement and (4) key management. Each controller uses MapReduce to schedule and distribute computation to cluster machines. We describe the file upload and encryption process next. Other controllers implement similar workflows.

A. *File Encryption Controller*

The implementation of the *File Encryption Controller* is shown in Figure 4. After the file is uploaded and stored in HDFS, the encryption requests are en-queued. The queue is periodically scanned to form a to-be-encrypted-file list, which is also stored as an HDFS file, and serves as the input to the MapReduce job. The Hadoop runtime distributes the encryption tasks to available machines through MapReduce. Each task receives a subset of to-be-encrypted files (the MapReduce job input) as the commands to perform file encryption. For each file to be encrypted, the task downloads an uploaded file (in plain-text) from the HDFS to the local machine, generates a key to encrypt it, and then deposits the encrypted file back to HDFS. The encryption key is stored in the encryption key store. The file encryption status is updated to allow encryption to be repeated in case of failures. More

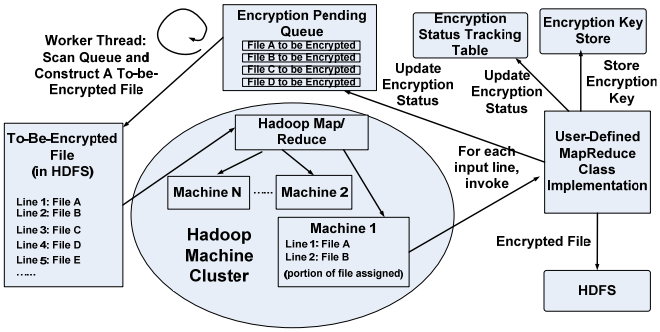

Figure 4. The distributed workflow based on Hadoop MapReduce for file encryption controller.

detailed discussion on failure recovery is provided in Section 5.

B. Concurrent and Batch Oriented Controller Execution

The task controllers submit their MapReduce jobs to the shared machine cluster concurrently, and are scheduled using the fair scheduler in Hadoop [14]. Each task controller is assigned its own unique pool. The weight assigned to each pool depends on how time sensitive the management task is. The fair scheduler relies on the weights to allocate MapReduce tasks and ensures that resources are distributed fairly between the task controllers.

MapReduce is inherently batch oriented, and therefore the service access APIs exposed by the architecture in Figure 3 are also batch oriented. Asynchronous processing is not an issue for archival solutions. To support real-time synchronous access for interactive applications such as browsing medical records, the Online File System in Figure 1 can cache original files or recently decrypted files. A slight modification of the service architecture is needed to support on-demand file retrieval with synchronous (rather than batch-oriented) file decryption.

IV. Encryption Key Store

The encryption key store is the most important component in our management service. This section presents the design of a highly secure and highly available encryption key store.

Having an encryption key store only hosted in the *service data center* where the retention management service is hosted, introduces a single point of failure and vulnerability. Instead, in our management service, encryption keys are partitioned into key fragments through polynomial secret sharing [23], and distributed to different key fragment stores at different data centers called *key distribution data centers*. No master keys are required to secure the key store.

In the encryption key store, each managed object is denoted by a unique Uniform Resource Identifier (URI). The key store is a key-value store that consists of a tuple with the object URI as the key, and the encryption key EK as the value, denoted as <URI, EK>. The encryption key EK_i for a file named by URI_i is partitioned into *n* key fragments, i.e., $EK_{i,1}$, $EK_{i,2}$, …, $EK_{i,n}$. Each key fragment is sent to one of *n* key distribution data centers, along with URI_i. That is, the *j*-th key distribution data center hosts the encryption key fragment store for the encryption key EK_i, denoted as $<URI_i, EK_{i,j}>$. The service data center can reconstruct the encryption key EK_i, based on a sufficiently large subset $\{EK_{i,1}, \ldots EK_{i,k}\}$ returned from *k* of the *n* key distribution data centers, where $k < n$. At the *j*-th key distribution data center, each pair of $<URI_i, EK_{i,j}>$ is stored in a scalable structured data store.

The encryption key store that employs polynomial secret sharing is both highly available and highly secure. Only *k* of *n* key fragments are necessary to reconstruct the key, even if some distribution data centers are down. An intruder will need access to at least *k* independent data centers to collect the necessary key fragments to reconstruct the key. On the other hand, if the intruder deliberately destroys some of the key fragments, a sufficient number of the key fragments from the non-compromised key distribution data centers allows successful key reconstruction.

An access management protocol is defined for the service data center to distribute (and retrieve) key fragments from the key distribution data centers. Standard web-based protocols such as those used in Amazon S3 [1] can be used to communicate between the service data center and the key distribution data centers. The request from the service data center to put (and retrieve) key fragments is signed with the secret access key granted to the service data center by the key distribution data center. A valid digital signature allows the key distribution data center to prove key ownership and thus grants key-related access to the service data center. Similarly, a public key based protocol can be used to sign and verify the request.

A. Secure Key Distribution

Secure key distribution between the service data center and key distribution data centers is shown in Figure 5. Message exchange between the service data center and the key distribution data centers can be achieved via a scalable message queue mechanism, or a remote procedure call (RPC) based protocol such as web services.

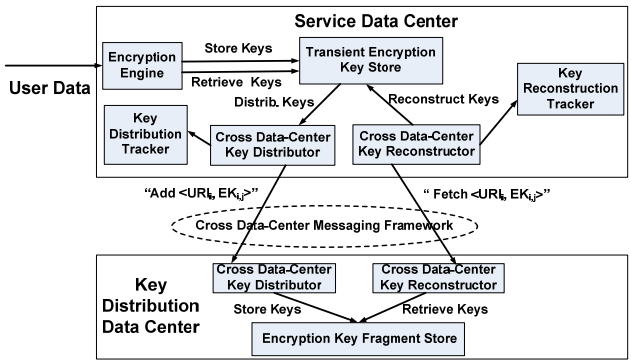

Figure 5. Encryption key partitioning and key fragment distribution.

In the service data center, once user data is encrypted, the encryption key is temporarily stored in the *Transient Encryption Key Store*. The *Cross Data-Center Key Distributor* takes the key that is newly inserted in this transient store and partitions it into key fragments. At the *j*-th key distribution data center, the Cross Data-Center Key Distributer receives the key distribution message and stores the fragment $EK_{i,j}$ in its key fragment store as $<URI_i, EK_{i,j}>$ for the data object with URI_i Once the key distribution to all data centers is successfully acknowledged, EK_i is removed from the Transient Encryption Key Store and subsequently, the user object (in plain-text) is deleted and only the encrypted object remains.

Due to machine or network failure, the message sent to the key distribution data centers may not always be acknowledged promptly, and key distribution messages may need to be retried until positive acknowledgements are received. Alternatively, it is possible to proceed with fewer fragments as along as at least enough acknowledgements have been received to enable the service data center to re-construct the key from the fragments that have been acknowledged. A batch processing task is invoked by the Cross Data-Center Key Distributor at the

service data center to scan the Key Distribution Tracker to perform key fragment distribution, determine whether the encryption keys have been successfully distributed, and retry distributions of unsuccessful key fragments. Such a key distribution batch processing task can be implemented with MapReduce and incorporated into the key management task controller shown in Figure 3.

B. Key Reconstruction

The *Cross Data Center Key Reconstructor* in the service data center issues the key reconstruction message to key distribution data centers that hold relevant key fragments. Once a sufficient number of key distribution data centers respond with the stored encryption key fragments, the service data center can reconstruct the encryption key. Because the communication delay between two data centers within the same geographical zone is small, to support fast key reconstruction, the key distribution data centers located within the same geographic zone as the service data center can be assigned to hold a sufficient number of key fragments to reconstruct an encryption key under normal conditions. However, to protect against natural disasters, a sufficient number of key fragments are also required to be distributed to key distribution data centers in other geographical regions, to facilitate key reconstruction only from the fragments held by cross-region key distribution data centers.

C. Key Deletion

The service data center receives a key deletion request either because the corresponding data's retention time expires, or because the data owner explicitly requests permanent data destruction. The request is stored in the *Cross-Data-Center Key Destructor* in the service data center and the request is returned. The key deletion request is then broadcast to all key distribution data centers to remove the key fragments that they hold. A batch processing task can be implemented with MapReduce and incorporated into the key management task controller to handle key destruction. When transient failures occur to the key distribution data centers, the key destructor will retry the unsuccessful key fragment deletions until a sufficient number of the fragments are successfully deleted.

V. Failure Recovery

The traditional data analyses conducted on a Hadoop cluster can tolerate both machine and task execution failures by restarting the Map or Reduce tasks on a different machine with the same file input. The output file is the only effect produced by the task execution. Such repetitive execution is straightforward because the analysis-oriented computing in Map or Reduce task is idempotent. That is, unchanged input always produces the same output.

In our policy management service, the situation is different. A Map or Reduce task associated with a policy management controller consists of multiple execution steps, each of which can touch multiple persistent stores and leave persistent states behind. The persistent states resulting from incomplete Map or Reduce tasks are side-effects. Correct task execution depends on persistent states recorded by the intermediate steps from the previously failed task execution. Failures that happen to one of the Map or Reduce execution steps can lead to unsuccessful file encryption. Failures can include (1) communication failure to backend structured data stores, (2) failure to local file systems; (3) communication failure to HDFS; and (4) crash of machines and processes.

In this section, we examine how our management task controllers can be designed to tolerate failures within MapReduce by using the File Encryption Controller shown in Figure 4 as an illustration example.

A. File Encryption Map and Reduce Execution Steps

In the distributed workflow shown in Figure 4, a file denoted as *F* is initially stored in the HDFS after being uploaded to our management service. *F-HDFS* denotes the file stored in HDFS and *F-Local* denotes the file stored on a local task execution machine. *F-Encrypted-Local* denotes the encrypted file stored on a local machine and *F-Encrypted-HDFS* denotes the encrypted file stored in the HDFS. The Map Task is implemented in the following steps:

(M1) Download *F-HDFS* to the scheduled task execution machine's local temp directory to become *F-Local*.

(M2) Encrypt *F-Local* with encryption key Key_F to produce *F-Encrypted-Local*; upload *F-Encrypted-Local* to HDFS as *F-Encrypted-HDFS*.

(M3) Compute Hash $Hash_F$ from *F-Local*.

(M4) Remove *F-Local* and *F-Encrypted-Local*.

(M5) Publish Key_F and $Hash_F$ to Encryption Key Store.

(M6) Perform integrity checking on *F*.

(M7) If the integrity of *F* is preserved, assign all *F-HDFS*'s file attributes to *F-Encrypted-HDFS*, then remove *F-HDFS*.

(M8) Update Status Checking Table with *F*'s encryption status;

(M9) Update Encryption Pending Queue with *F*'s encryption status.

The Map function does not produce output to the HDFS. The Reduce function does not handle management actions at all and thus is implemented as an *identity function* [14], which simply copies the supplied input as the processing output. The integrity checking routine called at Step M6 involves the steps of:

(M6.1) Download *F-Encrypted-HDFS* to a local file *F-Encrypted-Local*.

(M6.2) Retrieve Key_F and $Hash_F$ from Encryption Key Store.

(M6.3) Decrypt *F-Encrypted-Local* to *F-Decrypted-Local*.

(M6.4) Compute Hash (*F-Decrypted-Local*)

(M6.5) Remove *F-Encrypted-Local* and *F-Decrypted-Local*.

(M6.6) Return comparison result of Hash (*F-Decrypted-Local*) with $Hash_F$.

The integrity checking is designed to protect against data corruption due to transient errors. The original file can be safely removed from the persistent store, only if the stored

encrypted file *F-Encrypted-HDFS* can be successfully decrypted with data integrity guaranteed. This checking process slows down the overall file encryption, as it now involves both encryption and decryption.

B. Failure Recovery for File Encryption

The baseline mechanism to address failures is through retry of file encryption in the distributed workflow shown in Figure 4 that centers on the Encryption Pending Queue. If the encryption fails and no successful encryption status is updated, at the next scanning of the encryption pending queue, the file without successful encryption status is put back into the to-be-encrypted file. File encryption is then repeated at one of the cluster machines through MapReduce, until the maximum number of retries is exceeded and the failure status is recorded to the status tracking table.

Our failure recovery mechanism also takes advantage of the two features provided by the structured data store such as HBase. First, any data update to a given row is atomic and second, data updates to the same row are idempotent, because inherently the data store is a key-value store with a unique key for each row. Furthermore, a structured data store such as HBase is built to be highly available, and the store can self-heal should an internal failure occur. The internal failure states are invisible externally to the client.

We next focus only on communication failure to backend structured data stores to illustrate our failure recovery approach to deal with MapReduce task execution. Other failure situations can be handled similarly. As shown in Section 5.A, only the Map function is required for encryption related actions.

A Map task execution on a particular cluster machine can encounter a communication failure between the front-end servers and the back end structured data stores. A transient communication failure can be resolved by re-trying the data access request and eventually reaching the backend server [25]. Due to atomic and idempotent row-based update, updating a data store multiple times is not a problem. A permanent failure (e.g., due to the broken communication link) can be detected by the Hadoop job tracker through the established heartbeat protocol. As a result, the Map or Reduce task that has not finished will be re-launched on a different machine that is reachable by the Hadoop job tracker.

When the Map task is re-launched, the same file input that records the to-be-encrypted file list is re-submitted. Some of the listed files may have already been successfully encrypted. We need to determine whether a file has been successfully encrypted by checking both the encryption status tracking table and the encryption key table. Should either one of the two tables have not reported consistent positive confirmations, the file's encryption will need to re-start from Step M1. However, it is possible that the previous failure occurred during the execution of Step M7. If the query to HDFS shows that *F-HDFS* is removed and only *F-Encrypted-HDFS* is left, the Map task simply proceeds to its output step with a successful encryption status. Otherwise, the Map task starts from Step M1 and repeats the entire encryption with *F-HDFS*. A new encryption key will be created and be associated with *F-HDFS*.

Overall, for the Map task to address communication failure (and in fact, to also address the other failures identified earlier in this section), we can depend on the persistent states recorded at (a) the encryption status tracking table and the encryption key table (b) the HDFS regarding *F-HDFS* and *F-Encrypted-HDFS*, to determine which files need to be encrypted, and for which execution steps needs to be re-executed for individual file's encryption, in case the Map task is re-executed.

In general, in our policy management tasks, we can define all management functionality only in the Map functions. For each Map function *M* that supports an operation on an object (e.g., a file) *O*, we define an initial state (e.g., *F-HDFS* for file encryption), and a set of final states (e.g., updates to the status tracking table and encryption pending queue) for *O*. If operation to *O* succeeds, a *marker action* clears the initial state, before updating the final states. The marker action (e.g., to remove *F-HDFS*) needs to be atomic, whereas the update actions to all the final states do not need to be transactional. The implementation of *M* incorporates the following checks. If all final states of *O* are reached and consistent, *M* does not need to be repeated. Otherwise, *M* checks whether the initial state is cleared. If the initial state is cleared, *M* proceeds to the steps that only update the final states that represent successful operation on *O*, without performing object operation on *O*. The Map function *M* starts from the beginning if the initial state is not cleared.

The side-effects left from an incomplete Map or Reduce task's execution can include local transient files, e.g., *F-Encrypted-Local* and *F-Decrypted-Local*. Monitors installed on task execution machines can clean up such transient files in the background.

VI. Performance Measurements

We have prototyped the data retention management service shown in Figure 3 in a Linux cluster with 16 machines that are connected with a 10 Gb/sec network. Each machine has 8 cores and 32 GB RAM and runs with 64-bit Redhat Enterprise Linux. Hadoop 0.20.2 and HBase 0.20.3 are used in the prototype. The key management controller currently only handles key distribution within the same cluster, with the secret sharing scheme of <n=7, k=3>. That is, a key is partitioned into 7 fragments and 3 fragments are required for reconstruction. One cluster machine is configured as the master node of both Hadoop and HBase. The other 15 machines are configured to be the slave machine nodes. Each slave machine hosts a Hadoop data node and an HBase region server node. Each cluster machine is installed with Apache Tomcat 6.0 and the Axis web service framework as a web service front-end.

Each Hadoop slave machine also serves as a MapReduce task tracker node. The configuration is that at maximum, each task tracker node can support 6 concurrent tasks for encryption or decryption. That is, we allocated 6 out of 8 cores (i.e., 75% of computational capacity) per machine for encryption and decryption processing. As a result, there are a total of 90 cores within the cluster for file encryption and decryption.

Retention Policies Read, Write, Scan. We focused our performance measurement only on the backend service. The test clients run on the same machine cluster. Our first

measurement is the read/write throughput of data retention policies. The URI of each file is randomly generated. We pre-populated the HBase table with over 20 million policy objects, with each region server holding at least 4 regions, to ensure good load balancing among all the HBase region servers. In our cluster, we obtained 31,000 reads/sec and 15,000 writes/sec. The read performance is better than the write performance, as most of the reads are through the in-memory caches on the region servers [6, 15]. A 4GB heap size is allocated to each of the 15 HBase region servers. We estimate that the entire cluster should be able to hold 500 million policy objects in the combined in-memory caches.

We built a MapReduce-based scanner to scan the policy repository with about 125 million objects. The total scan took 1038 seconds. Based on this processing speed, the cluster will need 2.3 hours to scan 1 billion policy objects to determine which data objects have expired retention times.

File Encryption/Decryption. The second measurement that we did is on throughput of encryption and decryption. Our implementation used AES 256 encryption provided by the standard Sun JDK 1.6 distribution's crypto library. In our cluster, we achieved encryption throughput of 56 MB/sec and decryption throughput of 76 MB/sec. The measurement was taken after we uploaded 18,000 files, with each file having a fixed size of 2 MB. Thus we encrypted 36 GB in total. Our MapReduce Input Format [14] is designed so that the total encryption (or decryption) load contributed from all the 18,000 files is evenly distributed to each cluster machine and each machine has its maximum assigned number of cores (i.e., 6 cores) launched for encryption (or decryption). The encryption is slower than the decryption as our file encryption task involves file decryption to ensure data integrity.

To examine how encryption/decryption engine handles different file sizes, the measurement was repeated with a different fixed file size S_d (across all the uploaded files), that is, 0.25MB, 0.5 MB, 1 MB, 4 MB, in addition to 2 MB, while keeping the same total number of files, N (N=18,000), loaded in the cluster. To simulate heavy traffic from service front ends, the files were uploaded simultaneously by 5 concurrent test clients in each of the 16 cluster machines (i.e., each test client had 225 files to upload). Table 1 shows the total time spent on encryption (denoted as T_e) and decryption (denoted as T_d), and the measured throughput for encryption (C_e) and for decryption (C_d), given the input file size S_d. C_e is defined as the total file content to be uploaded (denoted as L, $L=N\times S_d$), over the encryption time measured (T_e). C_d is defined similarly.

Table 1. Encryption/Decryption throughput with respect to input file size.

	T_e(sec)	T_d(sec)	C_e(MB/sec)	C_d(MB/sec)
S_d=0.25MB	203	103	22.2	43.7
S_d=0.5MB	223	149	40.4	60.4
S_d=1MB	345	233	52.2	77.2
S_d=2MB	639	473	56.3	76.1
S_d=4MB	1274	883	56.5	81.5

Table 1 shows that when the input file size is small (less than 1 MB), the encryption or decryption throughput is much smaller than the one measured from input file sizes larger than 1 MB, indicating that our encryption/decryption engine is not efficient for files with small file sizes. As shown in Section 5.A, an encryption task execution consists of the steps that carry out crypto processing (i.e., encryption and hashing), and the steps that involve network file transfer to/from HDFS. Network file transfer is the overhead in throughput measurement and HDFS is designed for large files [13, 14]. When the files are small, such transfer overhead is significant, compared to the time for file encryption/decryption related operations. Table 1 also shows that after the file size reaches 1 MB, the encryption/decryption throughput becomes much less sensitive to the file size.

The performance measurement reported here is focused on the retention policy read/write throughput, retention policy enforcement, and data encryption/decryption throughput. The system availability related measurement requires further development and is not within the scope of this paper.

VII. Related work

Scalable Policy Management. In networked services and systems management, such as [24] that manages and enforces policies in Ponder [8], policies are distributed to policy management agents residing in networked computers and devices. Scalable policy management is achieved by having these distributed agents enforce the policies locally at each computer or device. In contrast, our management service is designed to manage data stored in a cloud environment. Specific to data retention policy, each physical machine does not assume the sole responsibility to control the lifetime of the data stored in such a distributed storage environment, and each machine can fail independently. Therefore the distributed management agent based approach is not viable in our environment.

A policy-based lifecycle management framework for the SAN-based file system is reported in [26], in which file-based lifecycle related policies such as data migration are enforced by the file-system's metadata server cluster. In our solution, the machine cluster in the policy management service serves the same role as the metadata server cluster. The key difference is that we use a common Hadoop and MapReduce framework to perform parallel policy enforcement actions with failure recovery being considered.

Enterprise-Level Compliance Solutions. HP TRIM [17] and Oracle's Universal Records Management [20] offer information lifecycle management (ILM) capabilities to help enterprises be regulatory compliant. They target single enterprises and offer no effective control over backup data, especially offline removable media. The archival solutions on fixed-content data (e.g., medical images) with long retention period (e.g., ten years and beyond) support data migration through a multi-tiered architecture (e.g., SAN, SCSI, and Tape) [5]. Overall, such solutions do not address effective deletion across multiple storage tiers when data retention period expires.

Compliance-Aware Cloud Services. Amazon.com Web Service (AWS) has little built-in data protection other than recommending that sensitive data should be encrypted. Applications that are developed with AWS, need to manage sensitive data within in-house environments and store only

encrypted data in AWS. Such applications are confronted with the challenge of building a scalable key management system. Salesforce.com supports disaster-recovery capabilities [22] mandated by regulations such as Sarbanes-Oxley and HIPAA, by backing up customer data to tapes in a separate data center, but not transporting tapes offsite. Our encryption-key based solution can help manage the data being backed up effectively.

Key Management. Vanish [12] relies on public, globally distributed hash tables (DHTs) to store the key fragments into DHTs, with the key fragments being constructed from the secret key sharing mechanism [23]. Our encryption key store is designed to be part of the management service that needs to regulatory compliant. All the key distribution data centers are under our control, and key operations are monitored. In contrast, the public DHTs that Vanish employs involve random machines from the Internet that join and leave irregularly. Moreover, Vanish relies on the un-guaranteed short-lived nature of the encryption key stored in public DHTs, and key operations cannot be monitored.

The Revocable Backup system [3] encrypts a file with a secret key. A master key encrypts the key file (in which all keys are stored) before the key file is backed up. File revocation is guaranteed by removing its encryption key and forgetting the master key of the key file from the previous backup run. Our system design adopts the same concept of using an encryption key to control the lifetime of a file, but focuses on managing the large collection of file objects in a centralized and scalable manner. Furthermore, our system does not need a master key to protect the encryption key store, and the key store is designed to be highly available and highly secure across different data centers, without the need to be backed up.

VIII. CONCLUSIONS

We have developed a data-retention policy management service, the first step towards building a scalable policy-aware policy management service that ultimately aims to support 10^{11} data objects and conform to compliance and regulatory policies. The service architecture is horizontally scalable. The data-retention service relies on a reliable and secure encryption key store that spans multiple data centers to store the encryption keys of data objects under management. The current system relies on Hadoop and MapReduce that are commonly available in a cloud-based environment to perform batch processing over a machine cluster for management tasks. We developed a state-aware failure recovery mechanism based on the recovery support from MapReduce to implement Map and Reduce functions for our policy management tasks. We prototyped the service and demonstrated it on a 16-node machine cluster. This cluster currently supports 56 MB/sec for encryption, 76 MB/sec for decryption, 31,000 retention policies/sec read and 15,000 retention policies/sec write.

We are currently extending the policy management service framework beyond data retention, to incorporate access control and privacy protection, and applying the framework to different application domains, such as healthcare, that involve sensitive data management to meet regulatory compliance requirements.

REFERENCES

[1] Amazon.com Simple Storage Service, http://aws.amazon.com/s3/.

[2] Z. Baird and J. Barksdale, "Implementing a trusted information sharing environment—using immutable audit logs to increase security, trust and accountability", Markle Foundation, 2006.

[3] D. Boneh and R. Lipton, "A revocable backup system," Proc. USENIX Security, pp. 91-96, 1996.

[4] P. L. Bradshaw, K. W. Brannon, T. Clark, K. Dahman, S. Doraiswamy, and L. Duyanovich, "Archive storage systems design for long-term storage of massive amounts of data," IBM J. Res. & Dev., Vol. 52, 2008.

[5] Bycast, http://www.netapp.com/us/products/storage-software/storagegrid/.

[6] F. Chang, J. Dean, G. Ghemawat, W.C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, "Bigtable: a distributed storage system for structured data," Proc. OSDI, pp. 205-218, 2006.

[7] Cloud Security Alliance, http://www.cloudsecurityalliance.org/csaguide.pdf.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder policy specification language," 2nd Int'l Workshop on Policies for Distributd Systems and Networks, pp. 18-38, 2001.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Comm. ACM, 51(1):107–113, 2008.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," SOSP '07, pp. 205-220.

[11] Directive 95/46/EC of the European Parliament and of the Council, http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf.

[12] R. Geambasu, T. Kohno, A. A. Levy, H. M. Levy, "Vanish: increasing data privacy with self-destructing data," USENIX Security, 2009.

[13] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google File System," Proc. SOSP '03, pp. 29-43, 2003.

[14] Hadoop, http://hadoop.apache.org/

[15] HBase, http://hadoop.apache.org/hbase/.

[16] HIPAA, http://www.hhs.gov/ocr/privacy/index.html.

[17] HP TRIM, http://h18006.www1.hp.com/products/software/im/governance_ediscovery/trim/index.html.

[18] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on automatic computing policies," 5th IEE Int'l Workshop on Policies for Distributed Systems and Networks, June 2004.

[19] K. L. Law and A. Saxena , "Scalable design of a policy –based management system and its performance," IEEE Commun. Mag., pp. 72-79, 2003.

[20] Oracle Universal Records Management, http://www.oracle.com/products/middleware/content-management/universal-records-management.html.

[21] PCI DSS (Version 1.2, Oct. 2008), https://www.pcisecuritystandards.org/security_standards/pci_dss_download.html.

[22] Salesforce.com Security and Compliance, http://www.salesforce.com/community/crm-best-practices/it-professionals/security-and-compliance.

[23] A. Shamir, "How to share a secret, " Comm. ACM, v.22, n.11, pp. 612--613, Nov. 1979.

[24] M. Sloman, "Policy driven management for distributed systems," J. Net. Sys. Mgmt. Vol. 2, No. 4, 1994, pp. 333-60.

[25] Varia, J.: Cloud Architectures. Available at http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf

[26] A. Verma, U. Sharma, J. Rubas, D. Pease, M. Kaplan, R. Jain, M. Devarakonda, and M. Beigi, "An architecture for lifecycle management in very large file systems," 2nd IEEE /13th Goddard Conf. on Mass Storage Systems and Technologies (MSST'05), April 2005.

[27] L. Wood, "Cloud computing and compliance: be careful up there," InfoWorld, 2009.