INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 4.295
(Volume 4, Issue 2)
Available online at: www.ijariit.com

A comprehensive view of encryption and automatic file updating system

Nirjhar Shukla
nirjhar183@gmail.com
Maharaja Agrasen Institute of Technology, Rohini, Delhi

ABSTRACT

Lately, the information that can be stored and analyzed in the petabyte scale datasets also known as Big Data have proved to be immensely utilitarian due to its innumerable applications such as analysis of user's choices which can be used in manipulation of options provided to a user by various firms and organizations. This if intended for the benefit of the user can be highly beneficial and fructuous. This Big Data is stored in storage places called as Cloud that have other kinds of data directly provided by users stored as well. Applications of these data directly provided by users usually entail storing large data chunks in certain designated locations by the users so that the user does not have to worry about managing data storages for the same. These applications have led to opening up of a whole new plethora of possibilities, trends and engagements. Unfortunately not all of these are legit and we can see malpractices cropping up every day due to manipulation of user's personal data for odious uses and ill-intentions. That being said, one must not undermine the cumulative storage size required for storing these magnanimous data. [Problems]: This means the major concerns for data storage in Cloud today are two. The first one is managing large data storage locations. The second one is safety of stored data from Machiavellian sources that unceasingly try to get hold of intricate data for malicious purposes. [Solution]: Therefore this research aims at finding a way out of these prevalent problems. This is achieved by aiming at the two major problems. The first one is dealt with using distributed file storage system which can be employed by an organization to store intricate personal data in their own computing systems. This would enable them the assurance of no data leakage by a foreign data storage firm that would have been managing their data on its own cloud. Using this distributed file system would also hold the organization accountable for any data breaches and hence the security of data could be managed in a better way. The second problem of security against malicious data intrusions is dealt with using encryption and decryption techniques while storing and retrieving data. The encryption-decryption technique utilized in this research is RSA. [Result]: By performing the aforementioned steps, the two major concerns that come with storing personal data on cloud can be handled efficiently.

Keywords: *Hadoop Distributed File System, RSA Algorithm, File Watcher System.*

1. INTRODUCTION

Cloud computing is a fast emerging model that has found its way into a number of applications which in turn make our everyday lives easier. The gadgets of today's world need not necessarily have huge storage spaces. With a given access to a cluster of storage units which may or may not be distributed, the worrisome task of storing large data is managed that too not at the expense of data on personal computing devices. This new paradigm is of great utility. But one has to understand that with this ease come a lot of trade-off which one must always pay heed to. Some of the issues that require our attention are following-

- One must always take note of the fact that the cloud services provided by a firm need not necessarily be stored in the same country as that of the organization storing the data. Thus the firm might peek into the intricate and private data of the firm under the pressure of host country. This is for the same reason that countries like China that had been long skeptical of policies and devious intentions of other major countries have only allowed usage of home-owned applications for day-to-day use such as search engines and messaging applications.
- The other issue that one must be aware of is the possibility of the Cloud firm trading personal data of an organization or a user to another organization which might be later on misused. Thus the security of the data stored in cloud is ensured only to a certain extent.

These problems are not ones that can be overlooked. Today there is an ongoing debate on breach of privacy of individuals especially in lesser developed countries like India. Most of it has to do with the well-known unethical access that sly corporations have to the critical datasets of individuals. Election campaigns are being run based on political inclination of individuals. Marketing strategies of corporations are greatly dependent on how users respond to certain logos and advertisements. The private photos and videos are getting leaked that were stored in the cloud. Other private details such as transaction history of a firm with another, the records of policies and decisions of governments are not safe today. All of these may or may not be related to the risk that cloud storage devices face today. Some of these risks might even be eye-opener for common man. But looking at it from the perspective of a proponent of information technology security, these breaches are detrimental to the very future of cloud storages especially as they have begun to outweigh the positives of these, the baring fact that we collectively have to face is that cloud security have to be managed at any cost. Security of the cloud can be maintained at many levels, ranging from first line of defense i.e making the users acquainted with the best practices to be followed while using information technology tools such as not storing private data in clouds, to more advance cloud securing methods such as using cryptograpy. Today we are provided with innumerable cryptography methods. Some are simple and easy to implement while others are more complex and harder to implement but on the other hand they provide greater security against malicious actions. One of the more advanced cryptographic techniques is RSA. This is a public key encryption and decryption method. Winding the head around RSA in order to breach the security layer is next to impossible for illegitimate sources. This is how the problem of security of cloud is solved. There still remains another unattended issue. This is that of the continuous confusion regarding whether to store extremely critical data in a cloud space or not. And if not, what place should be utilized for the same. The answer to this delirium lies in the Hadoop Distributed File System model which is available to us. The basic gist of the working of this system is explained in the following sections. HDFS of Hadoop Distributed File System is basically an architecture that enables us to utilize the storage space of a series of computer systems which are connected together as the storage space for multiple computing devices. This is to say that if n number of systems are connected and HDFS is enabled between these, one computing device might be able to store its own data in some other computing device's storage which must be designated beforehand as per the requisites of HDFS. If we try to analyze how this would help the corporations who are wary of the firms offering cloud storage spaces, we can see that since HDFS lets a corporation use the storage spaces of its own devices, hence the corporation alone will be responsible and accountable for any prospective data breaches. Hence the steps to subvert any probable security attacks will be dealt with the corporation itself in whatever way it sees fit. This makes sure that the possibility of Cloud manager deliberately providing data to other organizations for misuse is entirely vanquished.

2. HADOOP DISTRIBUTED FILE SYSTEM

HDFS is a Java-based file system that provides scalable and reliable data storage, and it was designed to span large clusters of commodity servers. HDFS has demonstrated production scalability of up to 200 PB of storage and a single cluster of 4500 servers, supporting close to a billion files and blocks. When that quantity and quality of enterprise data is available in HDFS, and YARN enables multiple data access applications to process it, Hadoop users can confidently answer questions that eluded previous data platforms.

HDFS is a scalable, fault-tolerant, distributed storage system that works closely with a wide variety of concurrent data access applications, coordinated by YARN. HDFS will "just work" under a variety of physical and systemic circumstances. By distributing storage and computation across many servers, the combined storage resource can grow linearly with demand while remaining economical at every amount of storage.

An HDFS cluster is comprised of a NameNode, which manages the cluster metadata, and DataNodes that store the data. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, or namespace and disk space quotas.

The file content is split into large blocks (typically 128 megabytes), and each block of the file is independently replicated at multiple DataNodes. The blocks are stored on the local file system on the DataNodes.

The Namenode actively monitors the number of replicas of a block. When a replica of a block is lost due to a DataNode failure or disk failure, the NameNode creates another replica of the block. The NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.

The NameNode does not directly send requests to DataNodes. It sends instructions to the DataNodes by replying to heartbeats sent by those DataNodes. The instructions include commands to:

- replicate blocks to other nodes,
- remove local block replicas,
- re-register and send an immediate block report, or
- shut down the node.

3. ANALYSIS OF HDFS

FileSystem in Hadoop is just an abstract class, which shields the underlying file system implementation, which is a bit of a Linux VFS flavor. The realization of each file system in Hadoop inherits the abstract class FileSystem, so the user only needs to operate the underlying file system through the FileSystem interface. There are many specific file system implementations in Hadoop, such as DistributedFileSystem and LocalFileSystem. How to tell Hadoop to use a specific file system implementation is discussed in this section.

Here we can do this by specifying fs.default.name in the Hadoop configuration file, which of course can also be explicitly specified in the program. Suppose we specify fs.default.name as hdfs: // localhost: 9000, where *hdfs* tells Hadoop to use DistributedFileSystem. Hadoop will extract hdfs from fs.default.name and go to the config file for fs. Hdfs .impl, which can be found in core-default.xml and has the value org.apache.hadoop.hdfs.DistributedFileSystem, as you can see Hadoop has been able to find out exactly what the file system implementation is to be created. Hadoop then reflexes to create class instances from class names and returns them to the user. Similarly if we specify the value of fs.default.name as file: /// then Hadoop will go to the config file for fs. *File* .impl, which is org.apache.hadoop.fs.LocalFileSystem. Of course, we can also specify many other values for fs.default.name to tell Hadoop what underlying file system implementation to use.

For HDFS we are through the DistributedFileSystem file system operations, where DistributedFileSystem is the client, and the server ran on the NameNode above. All operations that we issue via DistributedFileSystem are passed to the NameNode's server via rpc, and then the final result is returned. This is the rpc between client and server.

This section talks about how the client passes the user method call to DistributedFileSystem to the server. DistributedFileSystem can see the method call is passed to the instance variable dfs, this is an instance of DFSClient, this is a key point of rpc, the official document describes this class is

DFSClient can connect to a Hadoop Filesystem and perform basic file tasks. It uses the ClientProtocol to communicate with a NameNode daemon, and connects directly to DataNodes to read / write block data. Hadoop DFS users should obtain an instance of DistributedFileSystem, which uses DFSClient to handle filesystem tasks.

It can be seen that DistributedFileSystem hands over all file system operations to DFSClient for processing. Next we will see how DFSClient handles it.

DFSClient can be seen again turn the method call to the instance variable namenode, so we have to understand namenode and its associated rpcNamenode these two instance variables. For the DFSClient constructor, only the two most important things shown here are namenode and rpcNamenode respectively. To understand these two things first need to understand Java's Dynamic Proxy . We can see that rpcNamenode implements the ClientProtocol interface, which specifies all the methods a client can call to a server. According to the Dynamic Proxy mechanism we know that all of the method calls to rpcNamenode boil down to Invoker's invoke() method, this method is to transfer the client's calling information, such as calling methods, parameters, etc., through the socket to the server and get the result . The analysis of Invoker's invoke()method is not described here, and can be found at the link for more information. At this point the client's method call passed to the server successfully. Just look at DFSClient as long as the mkdirs() method of rpcNamenode is called, why namenode?

Watching directory for changes

Have you ever found yourself editing a file, using an IDE or another editor, and a dialog box appears to inform you that one of the open files has changed on the file system and needs to be reloaded? Or perhaps, like the NetBeans IDE, the application just quietly updates the file without notifying you. To implement this functionality, called *file change notification*, a program must be able to detect what is happening to the relevant directory on the file system. One way to do so is to poll the file system looking for changes, but this approach is inefficient. It does not scale to applications that have hundreds of open files or directories to monitor.

The java.nio.file package provides a file change notification API, called the Watch Service API. This API enables you to register a directory (or directories) with the watch service. When registering, you tell the service which types of events you are interested in: file creation, file deletion, or file modification. When the service detects an event of interest, it is forwarded to the registered process. The registered process has a thread (or a pool of threads) dedicated to watching for any events it has registered for. When an event comes in, it is handled as needed.

This section covers the following:

- Watch Service Overview
- Try It Out
- Creating a Watch Service and Registering for Events
- Processing Events
- Retrieving the File Name
- When to Use and Not Use This API

Watch Service Overview

The WatchService API is fairly low level, allowing you to customize it. You can use it as is, or you can choose to create a high-level API on top of this mechanism so that it is suited to your particular needs.

Here are the basic steps required to implement a watch service:

- Create a WatchService "watcher" for the file system.
- For each directory that you want monitored, register it with the watcher. When registering a directory, you specify the type of events for which you want notification. You receive a WatchKey instance for each directory that you register.

- Implement an infinite loop to wait for incoming events. When an event occurs, the key is signaled and placed into the watcher's queue.
- Retrieve the key from the watcher's queue. You can obtain the file name from the key.
- Retrieve each pending event for the key (there might be multiple events) and process as needed.
- Reset the key, and resume waiting for events.
- Close the service: The watch service exits when either the thread exits or when it is closed (by invoking its closed method).

WatchKeys are thread-safe and can be used with the java.nio.concurrent package. You can dedicate a thread pool to this effort.

Try It Out

Because this API is more advanced, try it out before proceeding. Save the WatchDir example to your computer, and compile it. Create a test directory that will be passed to the example. WatchDir uses a single thread to process all events, so it blocks keyboard input while waiting for events. Either run the program in a separate window, or in the background, as follows:

```
java WatchDir test &
```

Play with creating, deleting, and editing files in the test directory. When any of these events occurs, a message is printed to the console. When you have finished, delete the test directory and WatchDir exits. Or, if you prefer, you can manually kill the process.

You can also watch an entire file tree by specifying the -r option. When you specify -r, WatchDir walks the file tree, registering each directory with the watch service.

Creating a Watch Service and Registering for Events

The first step is to create a new WatchService by using the newWatchService method in the FileSystem class, as follows:

```
WatchService watcher = FileSystems.getDefault().newWatchService();
```

Next, register one or more objects with the watch service. Any object that implements the Watchable interface can be registered. The Path class implements the Watchable interface, so each directory to be monitored is registered as a Path object.

As with any Watchable, the Path class implements two register methods. This page uses the two-argument version, register(WatchService, WatchEvent.Kind<?>...). (The three-argument version takes a WatchEvent.Modifier, which is not currently implemented.)

When registering an object with the watch service, you specify the types of events that you want to monitor. The supported StandardWatchEventKinds event types follow:

- ENTRY_CREATE – A directory entry is created.
- ENTRY_DELETE – A directory entry is deleted.
- ENTRY_MODIFY – A directory entry is modified.
- OVERFLOW – Indicates that events might have been lost or discarded. You do not have to register for the OVERFLOW event to receive it.

The following code snippet shows how to register a Path instance for all three event types:

```
import static java.nio.file.StandardWatchEventKinds.*;

Path dir = ...;
try {
   WatchKey key = dir.register(watcher,
                          ENTRY_CREATE,
                          ENTRY_DELETE,
                          ENTRY_MODIFY);
} catch (IOException x) {
   System.err.println(x);
}
```

Processing Events

The order of events in an event processing loop follow:

a) Get a watch key. Three methods are provided:
   - poll – Returns a queued key, if available. Returns immediately with a null value, if unavailable.

   - poll(long, TimeUnit) – Returns a queued key, if one is available. If a queued key is not immediately available, the program waits until the specified time. The TimeUnitargument determines whether the specified time is nanoseconds, milliseconds, or some other unit of time.
   - take – Returns a queued key. If no queued key is available, this method waits.

b) Process the pending events for the key. You fetch the List of WatchEventsfrom the pollEvents method.
c) Retrieve the type of event by using the kind method. No matter what events the key has registered for, it is possible to receive an OVERFLOW event. You can choose to handle the overflow or ignore it, but you should test for it.
d) Retrieve the file name associated with the event. The file name is stored as the context of the event, so the context method is used to retrieve it.
e) After the events for the key have been processed, you need to put the key back into a ready state by invoking reset. If this method returns false, the key is no longer valid and the loop can exit. This step is very **important**. If you fail to invoke reset, this key will not receive any further events.

A watch key has a state. At any given time, its state might be one of the following:

- Ready indicates that the key is ready to accept events. When first created, a key is in the ready state.
- Signaled indicates that one or more events are queued. Once the key has been signaled, it is no longer in the ready state until the reset method is invoked.
- Invalid indicates that the key is no longer active. This state happens when one of the following events occurs:
  - The process explicitly cancels the key by using the cancel method.
  - The directory becomes inaccessible.
  - The watch service is closed.

Here is an example of an event processing loop. It is taken from the Email example, which watches a directory, waiting for new files to appear. When a new file becomes available, it is examined to determine if it is a text/plain file by using the probeContentType(Path) method. The intention is that text/plain files will be emailed to an alias, but that implementation detail is left to the reader.

Retrieving the File Name

The file name is retrieved from the event context. The Email example retrieves the file name with this code:

```
WatchEvent<Path> ev = (WatchEvent<Path>)event;
Path filename = ev.context();
```

When you compile the Email example, it generates the following error:

```
Note: Email.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This error is a result of the line of code that casts the WatchEvent<T> to a WatchEvent<Path>. The WatchDir example avoids this error by creating a utility cast method that suppresses the unchecked warning, as follows:

```
@SuppressWarnings("unchecked")
static <T> WatchEvent<T> cast(WatchEvent<?> event) {
   return (WatchEvent<Path>)event;
}
```

With the implementation of distributed file system, we must also get help of an insfrastructure that can inform the user about any changes that may occur in the stored data. This would ensure the detection of any unintended or malicious changes in the data. To better understand this concept, we need to take notice of the working of the distributed file system. As already discussed in detail, the HDFS lets a user on a computing device store his/her data on some other device that is designated as the location for the data storage in HDFS. Though the computers need to be connected together either via wired or wireless channel, the storage device might still end up performing some unintended actions on the dataset. To make sure that any change in the data gets the attention of the user, file watcher system is used. Actions such as creation, modification, deletion and duplication can be watched using this. To implement this we need to have following-

a) An object that would implement Watchable interface- we generally use Path class for this purpose.
b) A series of events which that we need to be under our surveillance- we can use StandardWatchEventKind to implement the WatchEvent.Kind<T>.
c) We also need an event modifier that can qualify the way in which Watchable will be registered with a Watchservice.
d) And finally we need a watcher that can watch some watchable, for example, some random watcher that can watch the File System for changes. The abstract class for this purpose is java.nio.file.WatchService but here the FileSystem object that can create a watcher for the FileSystem is used instead.

4. CONCLUSION

- Cloud computing extends beyond a single company or enterprise.
- To ensure the correctness of users' data in cloud data storage, an effective and flexible distributed scheme with explicit dynamic data support was proposed.

It is indeed that cloud computing can prove to be a boon in today's work environment hence this paper deals with data security issues related to cloud computing so that data centers can provide a good environment to keep data. The above mentioned scheme revolves around the problem of data security and with the help of encryption at client side and steganography at server side provides a highly secure model that will not only solve the issue of data safety but also simple in its implementation and hence usage. As per now the above mentioned scheme has been implemented using java. In future, the technique of image compression would be added to improve storage.