Implementation & Analysis of EFRS Technique for Intrusion Tolerance in Distributed Systems

Mr. A.B. Chougule[1], Mr. G.A. Patil[2]

[1]Department of Information Technology, Bharati Vidyapeeth's College of Engineering, Kolhapur, Maharashtra, India

[2]Department of Computer Science, D.Y. Patil College of Engineering, Kolhapur, Maharashtra, India

Abstract

This paper includes designing and implementing a system that uses encryption-fragmentation-replication-scattering for the purpose of developing secure and dependable data storage within a distributed system. The system will consist of one central node which is assumed to be trusted and multiple storage nodes. Data is collected at the central node, which is then encrypted followed by fragmentation. Data fragments then undergo a hash function to give unique hash value of each fragment. These fragments are then replicated and scattered over the network. Thus, the system continues to provide service even in case of failure of some storage nodes.

Keywords: *Intrusion, intrusion tolerance, encryption, fragmentation, replication, scattering*

1. Introduction

The static file storage systems used to store the files at single location i.e. on one server. This storage system is unreliable in case of failure of server since data will not be available if the storage location is unavailable due to some reason. In case of corruption of file, complete data is lost. To avoid such problems today distributed storage networks are used, which consist of many computers at different locations connected to each other via internet. However in such systems there is no enforced replication of data. So, if one of the machines is disconnected then the data will not be available.

Another major problem in case of such distributed storage networks is of intrusion. Intrusion has been defined as an entry without permission. It has been categorized in two different types, viz. active and passive intrusion. Active intrusion involves unauthorized alteration of data while passive intrusion is theft of data and possibly misuse of it. A few terms related to intrusion must be noted.

Intrusion Detection – It deals with discovering several types of malicious behaviors that can compromise the security and trust of a computer system or a network.

Intrusion Prevention – It deals with monitoring the network and/or system activities for malicious or unwanted behavior and then reacting in real time to block or prevent such activities.

Intrusion Tolerance – It assumes that there will be attacks made on the system and some of them will be successful. But it a ims to keep the system working despite of such attacks.

We aim at making the distributed storage systems more dependable and secure using the intrusion tolerance technique – EFRS. We define a few terms and make the following assumptions.

Terms

1. **C-Node** – A central server which is assumed to be a trusted storage. The system will consist of a single C-Node.

2. **Storage node** – A generic data storage server. The system may consist of several storage nodes which are responsible for storage of data fragments and scattering.

The rest of the paper is arranged as follows:
The Section 2 details the procedure of encryption-fragmentation-replication- scattering. Section 3 describes how the entire system was implemented. Section 4 gives the analysis made of the system performance and illustrates the results obtained. Section 5 gives the conclusions made as a result of the evaluation.

2. EFRS Procedure

This section describes the basics of EFRS technique.

2.1 Encryption and Fragmentation

Encryption of data is performed at the central server using AES encryption. After performing the encryption, the data is fragmented into several fragments, which are all of same size except the last fragment. In some cases, the last fragment might be smaller in size.

Encryption along with fragmentation is done considering two important reasons.

1. Since a single storage location will contain only a few fragments of data, the theft of single location is of no use to the intruder. To decrypt the data, all the fragments must be put in correct order.

2. Since fragmentation increases the security of data, a simpler and faster cipher can be used for encryption.

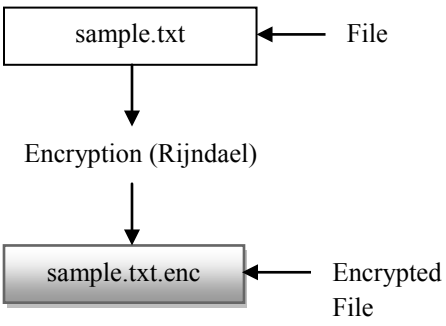

Fig. 1 Encryption

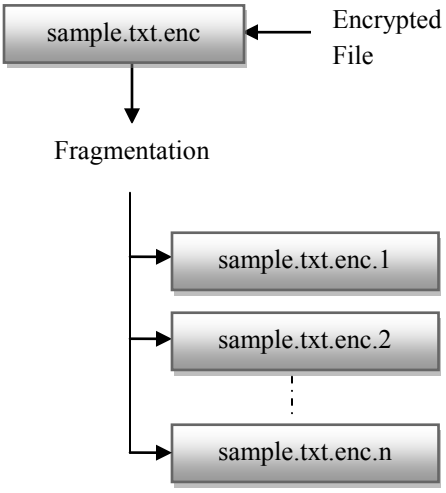

Fig. 2 Fragmentation

2.2 Hashing

The next step is to hash the fragments. Each fragment is acted upon by a hash function such as MD5, to obtain a unique hash value for every fragment. This hash value can later be used to confirm that the fragments have not been altered.

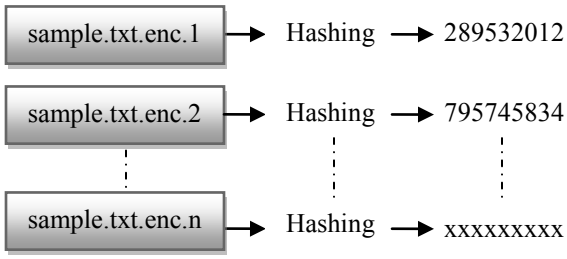

Fig. 3 Hashing

2.3 Replication

Now we assign a counter to each fragment. The counter (c) decides the number of copies of a fragment that will reside in the system. The counter is decremented at every stage as the fragment is forwarded from one storage location to another, thus creating multiple replicas of every fragment in the system.

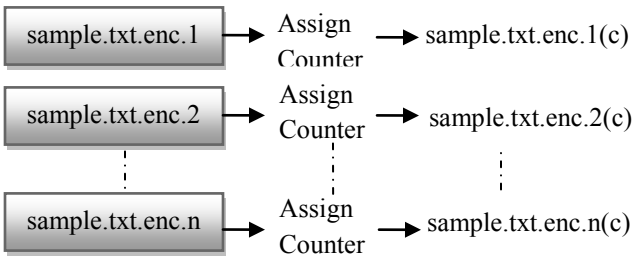

Fig. 4 Replication (assignment of counter)

Data availability is increased, due to the fragment replication within the system. To make a fragment unavailable the intruder would have to destroy as any sites as there were fragments in the system. The intruder would have no idea how many copies of the fragments were in the system, making it extremely difficult to know for certain that all the fragments had been destroyed. To affect the integrity of the data, an intruder would need to find and then modify all the replicas of a fragment stored in the system. This would require several intrusions. Even once they accomplished this, the cryptographic checksum of the fragment would be changed if even one byte to the fragment had been altered.

2.4 Distribution

The distribution is initiated by the C-node. It distributes the encrypted fragment of a d ata to different storage nodes making sure that not all the fragments of the data reside on a single storage node.

2.5 Scattering

In the scattering process, the counter assigned to each fragment is decremented at each storage node until it becomes zero. The storage node simply forwards the fragment if it has already stored another replica of it. The process of scattering increases the confidentiality, as a number of intrusions are required for an intruder to obtain all the fragments that make up a single piece of data.

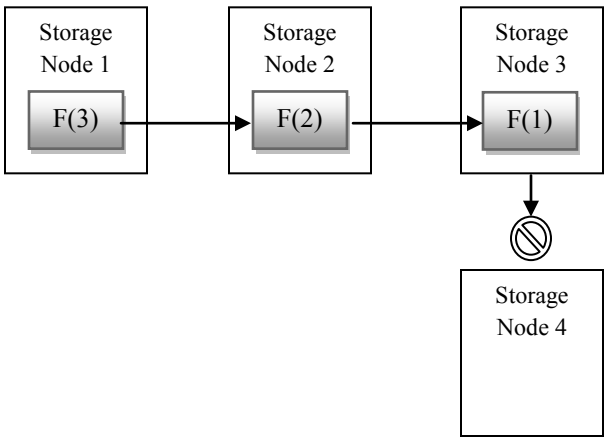

Fig. 5 Scattering

2.6 Retrieval

When the C-Node wishes to retrieve the fragments from within the system, a process that is almost the reverse of the fragmentation and scattering process takes place. For each fragment, the C-Node sends out a retrieve request to the network with the specific fragment name as part of the request. This request is forwarded from one storage node to another until the fragment is eventually found. The fragment then travels the reverse path until it finally reaches the C-Node. Figure 6 shows how the system would cope if a node failed. The C-Node is attempting to retrieve a fragment from the system. Consider, there were a copy of the fragment on Node4 and the request for that fragment has followed the path Node1→Node2→Node3. But Node3 has failed. Now the C-Node sends another request for the fragment to Node6. This request would then travel the path Node6→Node5→Node4. It then finds the fragment at Node4 and returns it to the C-Node by the path Node4→Node5→Node6.

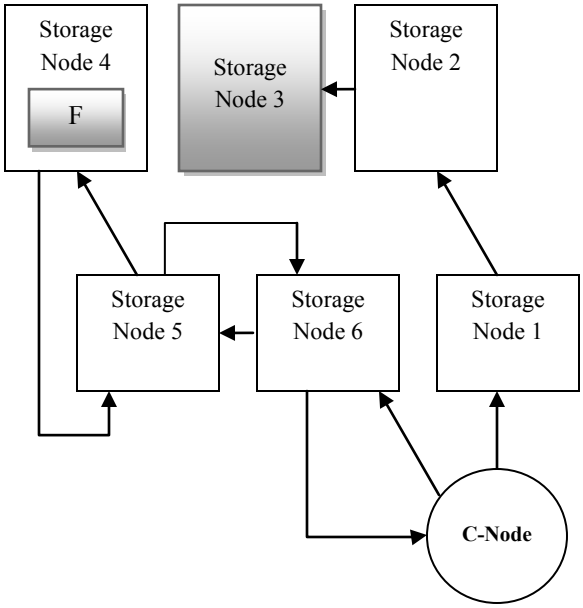

Fig. 6 Retrieval

2.7 Hash verification

When a fragment is retrieved successfully, the next step is to verify that the fragment has not been altered. The hash value for the fragment is re-calculated after retrieval and this value is compared with the original hash value as an integrity check. If the two hashes are the same, we can be sure that the fragment has not been altered in any way. This process is repeated for each fragment that arrives at C-Node until a viable copy of each fragment requested from the network has been obtained.

2.8 Data re-construction

Once all the fragments have been successfully retrieved, they are arranged into their original order and joined. Now we have the encrypted version of the data. The decryption is then performed on it, to obtain the original data.

3. Implementation

This section describes how the system is implemented to produce a d ependable and secure way to store data within a distributed system. The implementation has been divided into two parts: C-Node and Storage Node. These two components were

implemented separately and then integrated using .NET Remoting technology.

3.1 .NET Remoting

.NET remoting provides an abstract approach to interprocess communication that separates the remotable object from a specific client or server application domain and from a specific mechanism of communication. It is flexible and easily customizable. The remoting system assumes no particular application model.

3.2 C-Node

The functionality of the C-Node has been divided into several modules, which are as follows.

Encryption/Decryption

The data encryption within the C-Node is performed using the Rijndael algorithm. Rijndael is categorized under Advanced Encryption Standard (AES). AES provides greater security than its predecessor, DES and is also faster. It combines security, performance, efficiency, ease of implementation and flexibility and has low memory requirements. We have used 128bit block size and same key size for encryption/decryption. The methods used for encryption/decryption have the signatures as follows,

```
Boolean EncryptFile(String fileName,
String targetFileName, String sKey)
Boolean DecryptFile(String fileName,
String targetFileName, String sKey)
```

Fragmentation/Joining

The fragmentation function takes the fragment size `FragSize` and destination directory as the inputs. It forms a data block of `FragSize` KB in a single iteration and saves it as a fragment. This process is repeated until there is no more data to be processed. The joining function works in similar manner but in opposite way. The takes the directory name as input where the fragments are stored. It processes each fragment in a sequence and appends it to form a single data block. The following are the function declarations.

```
bool Fragment(string FileName, int
FragSize,string DirName)
void Join(string FileName, string
DirName)
```

Hashing

The hashing of the fragments is done by using MD5 hashing algorithm. **MD5 (Message-Digest algorithm 5)** is a widely used cryptographic hash function with a 128-bit hash value. MD5 processes a variable-length message into a fixed-length output of 128 bits. The function signature is as given below.

```
string GetHash(string FileName)
```

3.3 Storage Node

The storage node has been implemented in several modules. The storage node should handle three types of requests – store, retrieve and status check.

Fragment structure

Storage node defines a structure `Fragment`. It holds all the information related to the fragment and the fragment data. Every fragment has name, sequence number, counter, source location identifier and the actual data. The structure `Fragment` is defined with following members.

```
string sourceIP;
string name;
byte[] data;
int counter;
int SeqNo;
```

Request Handler

The request handler is the core component of the storage node implementation. It handles the requests for storage of fragments, retrieval and the status requests. The `RequestHandler` class implements an interface `IStorageNode`. Objects of this class are remotable i.e. they can be accessed through other processes which may reside on same machine or on a different machine.

The `IStorageNode` declares the three methods:

```
1. string CheckStatus();
2. GetFragment(int ReqID, string
FragName, string SourceIP);
3. StoreFragment(Fragment Frag);
```

Despite of these three methods, the interface also declares some additional methods to achieve some supporting tasks.

Storage request

For the process of handling the requests for storage of a fragment, the storage node first checks if the fragment has been already stored using the function, `isFragAlreadyStored()`. If it has not stored the fragment already, it will store the fragment, decrement the counter and then – if the counter has not reached zero – forward the request. If the fragment is already stored, then the storage node will just forward the request without storing it. The storage will only know one link back to the chain to the originator of the request. The storage node will also check to see if all the fragments of same data are being stored on it and if so, it will avoid such case by merely forwarding the fragment to some other storage node. Thus, all the fragments of the data will never reside on a single storage node, which adds to the security of the system

.

Retrieval request

If the message that the storage node receives is a retrieval request, the storage node first checks the database to see if it has the fragment stored. If it does, it will send the fragment to the node that requested it. If the storage node does not have the fragment stored in its database, it will record which storage node requested the fragment and then forward on the request to other storage node. Therefore, if a n ode further in the system does have the fragment stored then it can be passed back through the system following the reverse path. Each retrieval request will have a unique request identifier. Whenever a storage node receives a r etrieval request, it checks the identifier to determine if it has already processed the same request and if so, it will not forward the request this time, thus avoiding flooding of requests.

Status request

The status request is used to check if the storage node is working. When the storage node receives a status request, it returns an "OK" message back to the requester. Thus, if the storage node has failed, the requester will not receive an OK message within the given timeout period and will have to select some other storage node for further processing.

4. Evaluation

The EFRS system was tested on a wired local area network with speed 1Gbps. Each node in the testing environment had the following configuration:

1. Operating system – Microsoft Windows XP
2. Processor – Intel Pentium IV
3. RAM – 1GB DDR

The tests were carried out keeping some of the parameters constant such as the number of storage nodes, number of replicas and the size of the fragments. Three different experiments were carried out by taking a different fragment size for each of them. The values of constant parameters were as follows:

1. File size – 22959KB
2. No. of replicas – 3
3. Total no. of storage nodes – 6

Experiment 1:
Fragment size – 1024KB
Initially all the storage nodes were kept in working state and the time required for the retrieval of the fragments was measured. Same process was repeated three more times by increasing the failed nodes.

Table 1. Experiment 1: No. of storage failed nodes vs. time required for retrieval

No. of Failed Nodes	Time Required (ms)
0	13250
1	15328
2	17750
3	19110
4	20752
5	23661

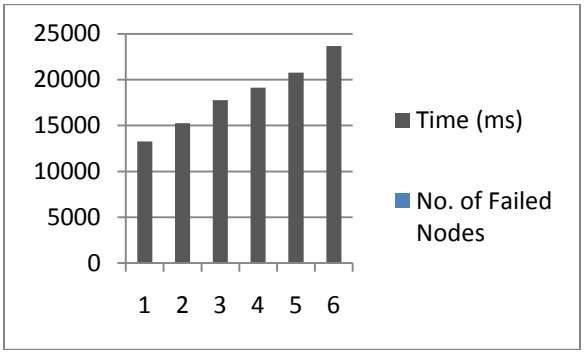

Fig. 6 Experiment 1: No. of storage failed nodes vs. time required for retrieval

Experiment 2:
Fragment size – 5120KB
Same experiment was performed with fragment size 5120KB. We can observe that the increase in the size of fragment has considerably reduced the required time.

Table 2. Experiment 2: No. of storage failed nodes vs. time required for retrieval

No. of Failed Nodes	Time Required (ms)
0	4078
1	5484
2	9703
3	11310
4	13245
5	15548

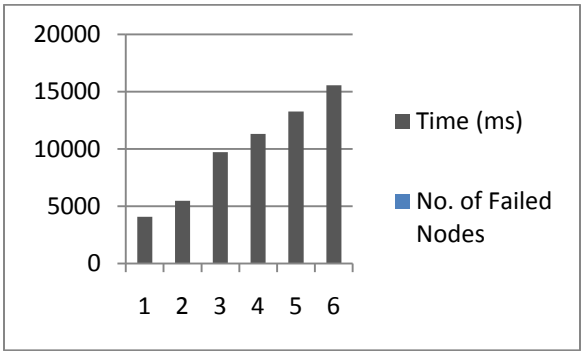

Fig. 7 Experiment 2: No. of storage failed nodes vs. time required for retrieval

Experiment 3:
Fragment size – 7168KB
Same experiment was performed with fragment size 7168 KB. We can observe here that the rate of time reduction has declined.

Table 3. Experiment 3: No. of storage failed nodes vs. time required for retrieval

No. of Failed Nodes	Time Required (ms)
0	4703
1	8359
2	12375
3	15430
4	19740
5	22584

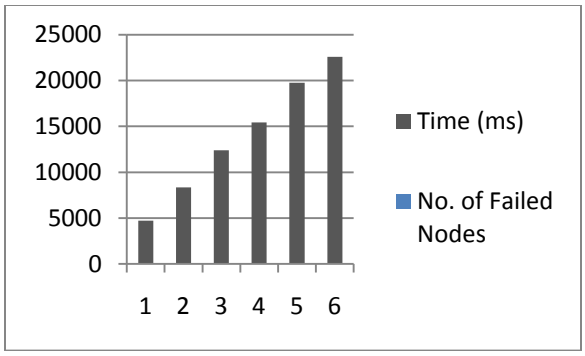

Fig. 7 Experiment 3: No. of storage failed nodes vs. time required for retrieval

5. Conclusion

The EFRS system provides a secure and dependable way to store data in a distributed environment. The system stores the data in form of fragments. Each fragment contains only a chunk of data. Moreover, all the fragments can never be found on a single storage node. So, if some intruder succeeds in retrieving some of the fragments, the attack can be of no use. Even if the intruder somehow manages to obtain all the fragments, the joining of fragments and decryption is an impractical task. Thus, the system tolerates passive intrusion.

The problem of active intrusion has been handled using the hash verification. The system verifies the hash value of each fragment as it is retrieved. If the hash value mismatches then the fragment is rejected and the system tries to find some other replica of it. In this way, even if some of the fragments are altered by the intruder, the system tolerates the attack and continues to provide service.

EFRS also addresses fault tolerance by using the same mechanism as in case of active intrusion attacks. The enforced replication of the fragments allows the system to keep working even in case of failure of some storage nodes.

Some limitations have also been observed in case of EFRS. The encryption, fragmentation and hashing is done on a s ingle central server. Failure of central node can result into loss of crucial data such as encryption keys.

References

[1] Victoria Spurrett, University of Kent, UK, "*Intrusion tolerance in dynamic distributed systems*"
[2] Y. Deswarte, L. Blain, J.-C. Fabre. *Intrusion Tolerance in Distributed Computing Systems*. Proceedings of the IEEE Symposium on Security and Privacy.
[3] J.C. Fabre, Y. Deswarte, B. Randell *Designing Secure and R eliable Applications using Fragmentation- Redundancy-Scattering: an O bject-Orientated Approach* in 1st European Dependable Computing Conference