Design and Implementation of a Scalable, Fault tolerant, Heterogeneous and Secured Distributed Storage Framework

Jerre Louis C.L.J. (Postgraduate Student)
Department of Computer Science
SSN College of Engineering
jerrelouis@gmail.com

Aravindan C.
Department of Computer Science
SSN College of Engineering
aravindanc@ssn.edu.in

Abstract

The design and implementation of a scalable, fault tolerant, heterogeneous and secured distributed storage framework named SCAVENGER, which aggregates the free storage spaces of the personal computers, is presented in this paper. The datasets are fragmented and stored with enough replication in different systems. The design includes a SCAVENGER protocol suite to monitor the activities like joining of new system, data storage, data retrieval and system failures. Multiple managers are used to make the design scalable. A synchronization protocol is designed to synchronize the data among the multiple managers. A model for fault tolerance is also designed based on the parameters like blackout time, overlap time, latency, trust, load, replication and redundancy. This is a minimization model which will be the fitness function for genetic algorithm to make decision upon the fragment size, the location of storage and the number of replications.

1. Introduction

The personal desktops are usually equipped with limited disk space. There are many personal desktops within the campuses whose storage spaces are unused. An advantageous and economical alternative to store large dataset which cannot be accommodated in a personal desktop is to bind the collective storage potential of individual personal desktops. These unused spaces can be aggregated and can be treated as a storage framework.

SCAVENGER is a distributed storage framework, which aggregates the free storage spaces available in the personal desktops. The personal desktops that donate some amount of its unused storage space are called benefactors; the controller of this framework is called manager and the users who either store or retrieve or delete the dataset are called client. The manager keeps track of all the meta-data information like the benefactor registration details, and location of the stored dataset. Although there exists work on desktop storage aggregation, SCAVENGER addresses the issues like scalability, fault tolerance, heterogeneity and security.

Multiple managers are used in the SCAVENGER framework to support thousands of desktop computers without any bottleneck in the framework. Fault Tolerance is achieved by replication. Choosing an arbitrary number of replications could waste the time for storing the dataset, network bandwidth, and the storage space of the benefactor systems. The need for having a fault tolerant system with minimal number of replications is required. In addition to the requirement of having a minimal number of replication, the datasets are stored onto the benefactors; the failure of these benefactors should not hinder the retrieval of the stored fragment. To fulfill the above two requirements an effective and efficient fault tolerant model is designed.

The rest of the paper presents the architecture and the design of the SCAVENGER framework. Section 2 discusses related work, Section 3 presents the architecture, Section 4 presents the issues tackled by the SCAVENGER framework, Section 5 presents the implementation details and Section 6 presents the results.

2. Related Works

The FreeLoader[12] closely resembles this work. However, SCAVENGER differs from FreeLoader in the architecture and storage of data onto the benefactors. What separates SCAVENGER from the other projects is its unique combination of features like security, design of genetic algorithm, design of protocol suite and usage of multiple managers.

There are many networked and distributed file systems exists some of them are NFS[9], LOCUS[11], CODA[4], AFS[7] and GFarm[10]. SCAVENGER is a lightweight distributed storage framework and not a file system. There are many P2P systems exists some of them are Kosha[1], Kazaa[13], PAST[6], Freenet[2] and BitTorrent[5]. SCAVENGER is not a P2P as well. The detailed related works can be referred from our previous paper [3].

3. Architecture and Design

The actors in the SCAVENGER architecture are the Client, the Manager and the Benefactors. The dataset is stored in the following manner. The client streams the dataset to the manager, upon receiving the size of the dataset the manager starts the genetic algorithm (GA) which decides about the number of replication, the number of fragments and the location of storage. The manager fragments the dataset, digitally signs, compresses and distributes the fragment to the corresponding benefactor. The receiving benefactor sends the received fragment to two other benefactors and so on. The retrieval of the dataset is done in the following manner. The client requests the stored dataset, the manager retrieves it from the benefactors and streams it to the requested client. The client may delete the stored dataset by providing the dataset name. All these operations are performed securely. The security is taken care by the protocol suite.

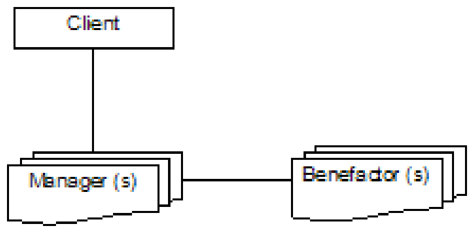


Figure 1. SCAVENGER Architecture

The rationale in designing this architecture is to have a framework that has no bottleneck and to reduce the burden on the client. A protocol suite is designed to monitor the joining of the new benefactor, to get the pulse of the benefactor, data storage, data retrieval, benefactor failures, benefactor leaving the framework and a synchronization protocol to synchronize the data among the managers. The designs of the protocols are discussed below and each of the protocol has been verified using the LTSA [8] tool.

3.1 SCAVENGER PROTOCOL SUITE

3.1.1 Joining of a new benefactor

The benefactor generates a public and private key and then join the manager by providing its public key (EUb), donated storage space, the location where the fragment is to be stored and its IP address. Then the manager validates if the donated storage space is available in that benefactor. The manager sends the acceptance by sending its public key (EUm) if the information provided by the benefactor is valid. The values given by the benefactor and its key are then stored in a database.

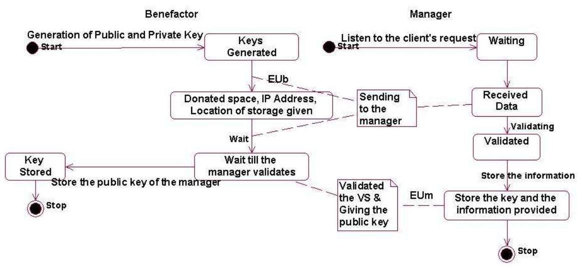

Benefactor
Manager
Generation of Public and Private Key
Start
Keys Generated
Listen to the client's request
Start
Waiting
EUb
Donated space, IP Address, Location of storage given
Sending to the manager
Received Data
Validating
Wait
Validated
Key Stored
Wait till the manager validates
Store the public key of the manager
Store the information
Validated the VS & Giving the public key
EUm
Store the key and the information provided
Stop
Stop

Figure 2. Benefactor joining

3.1.2 Pulse of the benefactor

The manager checks the pulse of the benefactor at regular intervals. In order to show the authenticity of the manager, it signs the probing request (SIGm). The benefactor responds by signing its response (SIGb).

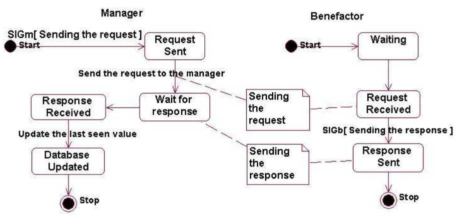


Figure 3. Checking the pulse

3.1.3 Data Storage

The client generates a session key (Ks) and sends it to the manager by encrypting it with the manager's public key (EUm). The manager acknowledges the client. The client streams the dataset to its manager by encrypting it with the session key. The manager decrypts the dataset, fragment the dataset, digitally signs that fragment, compress that fragment and then stream it to the benefactor by signing the data (SIGm). The benefactors are chosen for storing the fragments according to the decision given by the GA. The corresponding book keeping is done.

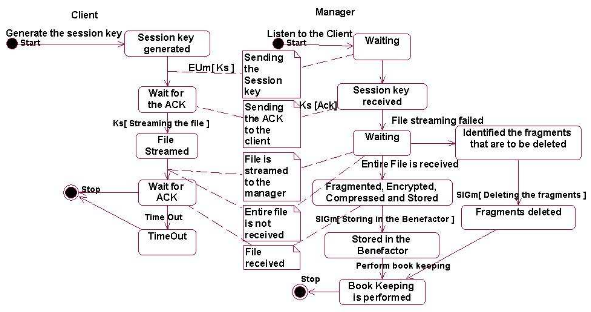


Figure 4. Data storage

3.1.4 Data Retrieval

The client generates a session key (Ks) which is given to the manager by encrypting it with the manager's public key (EUm). The manager acknowledges the client. The manager requests the corresponding benefactors by signing the request and retrieve the fragment, decompresses it, verify its signature and then stream it back to the client by signing the file fragment.

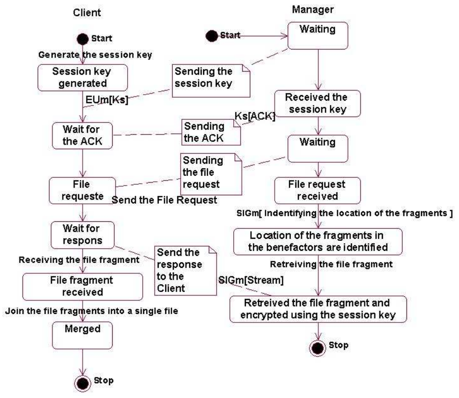


Figure 5. Data retrieval

3.1.5 Benefactor taking a planned leave

The benefactor may take a planned or unplanned leave from the framework. In the case of planned leave, the benefactor sends the request of leaving to the manager by digitally signing it. The manager move the contents from that benefactor to a new benefactor and the corresponding book keeping is done.

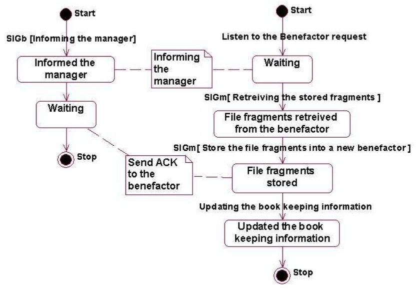

Start
Start
SIGb [Informing the manager]
Listen to the Benefactor request
Informed the manager
Informing the manager
Waiting
Waiting
SIGm[Retreiving the stored fragments]
File fragments retreived from the benefactor
Stop
SIGm[Store the file fragments into a new benefactor]
Send ACK to the benefactor
File fragments stored
Updating the book keeping information
Updated the book keeping information
Stop

Figure 6. Benefactor leaving

3.1.6 New Manager Joining

The administrator of the SCAVENGER framework may deploy any number of managers according to the requirement. If a system is to be made as a manager, it needs to contact any of the existing manager(s). The request is sent to the existing manager by the benefactor digitally signing (SIGb)the request. Once the request is made, an existing manager gives out the meta-data information and the keys of the benefactor systems by digitally signing it (SIGm). Once the meta-data information and the keys are received, the new manager acknowledges that manager. Upon the acknowledgment the existing manager informs all the benefactors about the joining of the new manager.

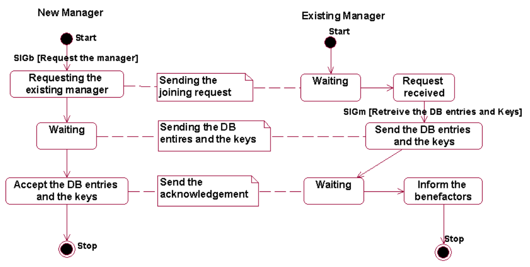

New Manager
Start
SIGb [Request the manager]
Requesting the existing manager
Sending the joining request
Waiting
Sending the DB entires and the keys
Accept the DB entries and the keys
Send the acknowledgement
Stop
Existing Manager
Start
Waiting
Request received
SIGm [Retreive the DB entries and Keys]
Send the DB entries and the keys
Waiting
Inform the benefactors
Stop

Figure 7. New manager joining

3.1.7 Manager Synchronization

The benefactor may either register / store / retrieve / delete the file fragments with / in / from the SCAVENGER framework. Once a manager receives any of the requests, it informs all the managers in the framework by digitally signing (SIGm) about the transactions made between that manager and the benefactor.

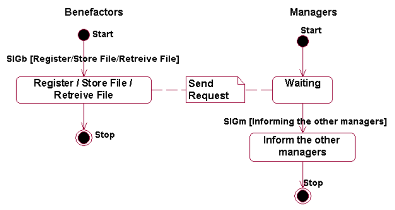

Benefactors
Start
SIGb [Register/Store File/Retreive File]
Register / Store File / Retreive File
Stop
Send Request
Managers
Start
Waiting
SIGm [Informing the other managers]
Inform the other managers
Stop

Figure 8. Manager Synchronization

4. Issues Addressed

SCAVENGER addresses the issues like fault tolerance, scalability, heterogeneous and security.

4.1 Fault Tolerance

A balance between the number of replication and fault tolerance is required, so an effective and efficient fault tolerant model is designed based upon the parameters like blackout time (BT), overlap time (OT), latency (Lat), replication (Rep), redundancy (RT), trust and load. Blackout time is the time during which the fragment is not available in any of the system. Overlap time is the time during which the same file fragment is available in multiple benefactors. Latency is the time taken by the benefactor to respond the manager. Replication is the number of copies made and stored in the benefactors. Redundancy is the number of times the same fragment is stored in a machine. Trust is the positive value given to the benefactor, if the benefactor is available at all time whenever the manager checks its pulse. The load is the number of fragments, average size of the fragment and space utilized in that system.

$$\sum_{i=1}^{f}[BT\_Term + OT\_Term + Replication\_Term] + \sum_{j=1}^{b}[Latency\_Term + Trust\_Term + Load\_Term + Redundant\_Term]$$

Where f is the number of fragments and b is the number of benefactor systems where the fragments are stored.

$$BT\_Term = BT\_Weight * BT$$
$$OT\_Term = OT\_Weight * OT$$
$$Replication\_Term = Rep$$
$$Latency\_Term = Lat$$
$$Trust\_Term = (Trust)^{-1}$$
$$Redundant\_Term = RT\_Weight * RT$$
$$Load\_Term = log(No.ofFrag) + log(AvgFragSize) + ((SpaceUsed/TotalDonSpace) * 100)^{-1}$$

This minimization model will be the fitness function for GA.

5. Implementation

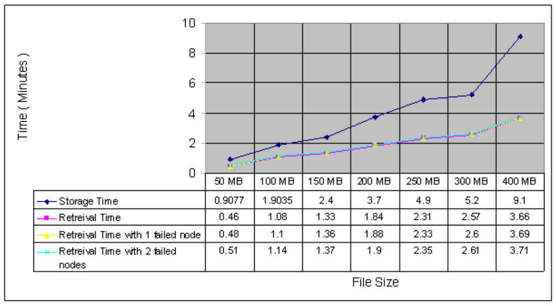
All the three programs (manager, benefactor and client) in this framework is developed using Java with Derby as the database. The implementation of GA is crucial in this framework as very quick decisions are to be made upon the numbers of replications, number of fragments and a place for storing these fragments. A solution in the GA is represented using a triplet $< m, n, \{BenefactorList\}\ m * n >$. Where m is the number of replications and n is number of fragments. The initial population is generated based upon the available time, available space, redundancy and response time of the benefactor. The fault tolerance model discussed earlier will the fitness function for GA. The selection of parents are done by the Roulette wheel method. A cross over point is chosen at random and the cross over is preformed on the chosen parents to form a new offspring. The newly generated offspring with least fitness value will replace the existing population to form a new generation. New generations are generated till the satisfying condition is met.

6. Results

This section presents the results of our prototype implementation. The results were taken when streaming a sample dataset of the size between 50 MB and 400 MB. The time taken for storing the dataset, retrieving the dataset, retrieval time of the dataset with a failure of 1 node and retrieval time of the dataset with a failure of 2 nodes is depicted in the form of graph (Figure 9). The results were generated with 15 benefactors that were well scattered within the Computer Science and Engineering department. The testbed configuration is given in Table 1.

SCAVENGER	15 Systems (15 Benefactors, 2 managers and 1 client).
Operating Systems	Windows XP, Mac 10.4 and Red Hat Linux
Processor Speed	1.4 MHz - 2.4 GHz
Storage Space	40 GB - 80 GB
Primary Memory	256 MB - 1 GB
Dataset	50 MB - 300 MB

Table 1. Testbed Configuration



	50 MB	100 MB	150 MB	200 MB	250 MB	300 MB	400 MB
Storage Time	0.9077	1.9035	2.4	3.7	4.9	5.2	9.1
Retreival Time	0.46	1.08	1.33	1.84	2.31	2.57	3.66
Retreival Time with 1 failed node	0.48	1.1	1.36	1.88	2.33	2.6	3.69
Retreival Time with 2 failed nodes	0.51	1.14	1.37	1.9	2.35	2.61	3.71

Figure 9. File size vs Time

References

[1] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. *In Proceedings of Supercomputing*, 2004.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *LNCS*, 2000.

[3] C.L.J. Jerre Louis and C. Aravindan. Design of a scalable, fault tolerant, heterogeneous and secured distributed storage framework. *In Proceedings of ICETET'08*, pages 1313–1316.

[4] CODA File System. *http://www.coda.cs.cmu.edu*.

[5] B. Cohen. Incentives build robustness in bittorrent. 2003.

[6] P. Druschel and A. Rowstron. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *In Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

[7] J. H. Howard. An overview of the andrew file system. 1998.

[8] Labelled Transition System Analyser. *http://www.doc.ic.ac.uk/ltsa/*.

[9] B. Nowicki. NFS: Network file system protocol specification. *Network Working Group RFC1094*, 1989.

[10] Osamu Tatebe et. al. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. *In Proceedings of CHEP'04*, September 2004.

[11] G. Popek and B. J. Walker. The locus distributed system architecture. *MIT Press*, 1985.

[12] S. Vazhkudai et al. Freeloader:scavenging desktop storage resources for scientific data. *In Proceedings of Supercomputing'05*.

[13] The kazaa media desktop. *http://www.kazaa.com/*.