File Replication

And

Performance Evaluation

Project Report Document

VERSION	1.0
DATE	6/07/2010
BY	Megha Gupta Megha Mathur Zhengning Xu

File Replication and Performance Analysis

1.0 Introduction

File replication is one of the most critical components of data-intensive computing environment. The need for file replication in timely fashion arises in various areas of data analysis such as high-energy physics, bio-informatics, climate modeling and astronomy. Companies like Cern and Google have data centers all across the world and are producing peta- bytes of data per year thus it is very important to have reliable and quick file replication.

1.1 Abstract

In this report we will be discussing about replication file system which is developed on FUSE. The file replication system will allow users to create replicas of files and create their backup on all network connected system. The file system will improve read performance by checking read latency of all system on network and reading from least latency system.

1.2 Objective

The objective of our term project is to design and implement a distributed file system for a small infrastructure in which files of different sizes will be replicated on different machines on the same network to increase the reliability and provide fault tolerance.

1.3 Problem Statement

Storing a file on just one machine does not guarantee data reliability and does not provide optimal read data efficiency.

1.4 Why is the project related to this class?

Since file systems is one of the most important aspects of Operating systems, integrity of the file system is of great importance in any industrial infrastructure.

1.5 Why is the other approach not good?

The other approach that we analyzed was of Google File System. It is appropriate for a big company like Google, but for a smaller setup, such an approach will turn out to be very expensive. It will not be feasible to implement it in a small setup.

1.6 How our approach is different?

Our approach is better in the following aspects –

1. In the Google File System, a single Master keeps track of the buddies. In our approach, Member Management runs on all the machines separately and each machine keeps track of its own buddies.

2. Different machines can join or leave the network whenever they feel like. In every 10 seconds, all the machines check to see that which other machines are available at the moment. Google File System does not provide any feature like this.

3. In Google File System, the read is always done from the local copy. Our approach improves the efficiency of the system by checking the read latency from each machine and makes sure that the file is loaded from the fastest node. In most cases, the local host is the fastest. But if due to some circumstances, the local host is busy at that time and cannot provide fast access to the file then our system will load the file from the fastest one.

4. The backup process in our approach is fully automatic. The backup copies are made at the same time as the local copy. Google File System uses a very different approach.

2.0 Theoretical basis and literature review

2.1 Definition of the problem

Data integrity and data reliability are very important issues in any industrial setup and most of the organization strive to achieve this. For this they plan to do backup and file replication. If it takes lot of time and money, it will not be of much help for the company so the important concern is to do backup quickly and with reliability.

One of the problems faced is reliable recovery of the files. When files storage is done on a single host system and due to unfavorable reasons the host crashes, recovery of data is impossible and this can lead to loss of important information resulting in loss of time and money for the companies/organization. Therefore, such systems cannot be trusted with important data storage.

Since File system is one of the most important aspects of the operating system, an optimal solution for this problem is needed.

Another challenge faced by file replication system is access time to data (data locality) and/or fault tolerance (data availability). Now days many organizations have globally distributed user communities and distributed data sites therefore it is very important to have fast and reliable access to data to make important and market savvy decisions.

2.2 Related research to solve the problem

We conducted research on similar file systems like Google File System and Pastiche – P2P backup file system. We also did some research on Efficient Management of Backups in Distributed Systems.

In our term project, we have tried to incorporate different aspects of these file systems and analyze the performance to compare it to the existing systems.

2.3 Our Solution to solve the problem

Our solution is to create a file replication system RFS, where a user can copy, save and update the files in a virtual directory and our replicated file system will make a copy of it on the network environment including the local machine. This will automate the process of backup and replication.

2.4 Where our solution is different from others?

Our solution is different in these areas –

1. Management of the buddies is decentralized on all the machines on the network in our approach. Every machine runs its own member management.

   In our research studies, this concept was centralized. This means that there was a single master who kept the records of the replicated files.

2. Read efficiency is another important aspect of our project. The read will always be done from the fastest buddy.

3. The backup is done automatically. Files are written on all the machines at the same time as the local copy.

2.5 Why our solution is better?

Our solution is better because we are trying to decrease the load on single machine by enabling every machine to keep track of their files and buddies. This approach will reduce the pressure from one machine and avoid bottleneck leading to better throughput. This will also reduce the cost of communicating with one master machine.

Our approach also improves the performance of the overall system by first evaluating the read latency of each machine and reading from the fastest one.

Also, explicit backups will not have to be done by the owner of the file. The files will be backed up automatically.

3.0 Hypothesis

Our aim is to improve the performance of retrieval and backup for the user files. We have done this by implementing simultaneous writes on different machines and reading from the machine with lowest latency.

4.0 Methodology

4.1 How to collect/generate input data?

We have used files that were already present on the system for backup. We have used files of different sizes to conduct our experiments and analyze our results.

4.2Algorithm Design

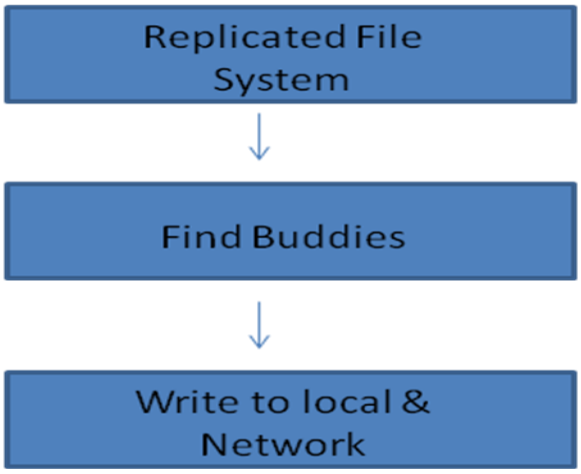
Modules implemented:

1. Member Management – This module performs the following functions:
    - Finding buddies (or hosts) on which the file can be replicated.

- Disk-space Management by deleting the files of non-active members.
- Adding new buddies to the network when they want to join.
- Removing buddies from the network when they want to leave.

2. File-Storage Subsystem – This module handles the actual writing (or replication) of the files on other machines.
   This module is responsible for maintaining a log of the files. It will record the metadata persistently and also serve as a logical time line that defines the order of concurrent operations. The file system can be recovered by replaying the operation log.

Figure1: Write Operation

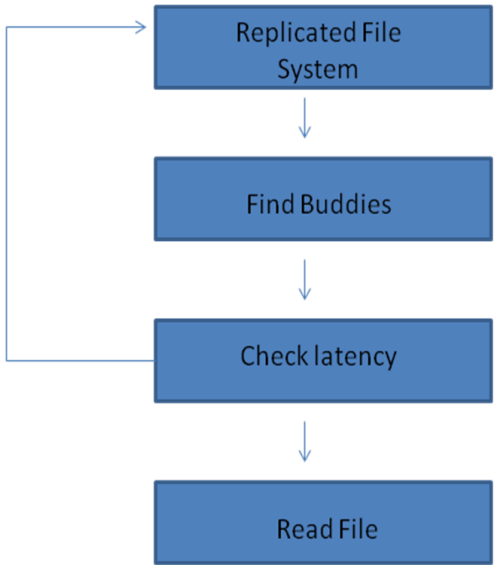


The Figure 1 explains the replication mechanism-

1. The File System asks the Member Management to find buddies to write the replicas on.
2. Once the buddies are found, an attempt is made to write (or replicate) the file on the local as well as remote machines.
3. If the write operation fails, another attempt is made to do the same.
4. If the write is successful, store the metadata of that write in the local machine in a list.

3. File-Retrieval Subsystem – This module is responsible for reading back the file from the buddies it is stored on for recovering the data. It monitors the latency between replicas and decides from where we are going to read the file.

Figure2: Read Back Operation



The Figure 2 explains the read back mechanism-

1. The file system first sends request to all the buddies to check the latency to read from each buddy.
2. All the buddies reply back with their latency.
3. The File system decides which buddy to get the file from.
4. The file is read from the buddy selected.

4.3 Language Used

We used C language for our project.

4.4Tools Used

We used File-system in User space (FUSE), an Open source software utility to create our own file system. FUSE is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create their own file systems. This is achieved by running file system code in user space while the FUSE module provides only a bridge to the actual kernel interfaces.

We have also used NFS – Network File System.

4.5 Prototype

Our prototype for replicated file system consists of member management, creating multiple copies of files and increasing the read efficiency. This will be done as follows –

1. When the new user joins the network and runs Replicated File System (RFS), all new and updated files are copied to the new user's directory from the oldest member's directory.
2. When the user wants to write a file, i.e. copy, update or save, the files will be first written in the virtual directory of user and then our replicated file system will copy it to the network environment.
3. When user wants to read the system will open the file from the machine with least latency.

4.6Output Generation

At the completion of Write, all the machines on the network will have a copy of the replicated file in their rfs directories.

4.7 How to test against hypothesis?

Our hypothesis is to increase performance of recovery and backup and we assume this can be done by multiple simultaneous writes and reading from the fastest node by measuring the latencies of all the nodes first. So to make write efficient our system creates multiple copies of file on various machine and for read performance when user issues read command then we will check the latency of each machine by copying a chunk of data in every machine and finding the latency.

5.0 Implementation

5.1 Design Document

RFS system federates all local file systems on different machines to provide a single virtual file system to applications. We summarize the design of the RFS into the following points:

a) RFS server architecture

Each machine has a RFS server running in the user-space. Multiple RFS servers constitute a virtual file system as a single member. A member can dynamically join and leave the virtual file system. Each RFS server mounts a directory as RFS mount point, e.g. /home/test/rfs and register it to the Fuse kernel module. Therefore every file operation, like open, create, write, read, close etc, to the RFS mount point will be forwarded to the RFS server and the RFS server will implement all file system related operation methods to embed the file replication logic.

Each RFS server maps the RFS mount point to a set of member directories, including a normal directory on local member and other NFS directories mounted from remote RFS member. Each read and write operation will be executed on the set of directories to achieve fault tolerance and automatic data backup.

b) Replicated write

To implement automatic data backup, for each data or metadata write, like write and mknod system call, RFS servers duplicates the operation by executing it on all member directories. By default the writes to all members are conducted sequentially. RFS also implement a parallel write alternative, in which for each member write, RFS use a separate Pthread to do the job. The execution flow of a replicated write in RFS is as follows:

1. Application initiates a write to a RFS file via standard file system call on a virtual file descriptor.
2. The fuse kernel module intercepts this write system call and forwards it to the local RFS server.
3. The RFS server iterates current member list. For each enabled member, RFS server finds out the corresponding real file descriptor, and conducts the write operation to that particular member. For each failed write operation, RFS will consider the target member is down and disable the member.
4. Once the write operation finishes for all enabled member, RFS check if there exists a successful member write. If yes, RFS return the bytes written. Otherwise it means all member writes fail, RFS returns error.

c) Fault-tolerant read

Another important design goal of RFS is to provide fault-tolerance of file operations. This is achieved by implementing fault-tolerant read operations. Basically the read operation will succeed as long as any of the members is available. The execution flow of a fault-tolerant read in RFS is as follows;

1. Application initiates a read to a RFS file via standard file system call on a virtual file descriptor
2. The fuse kernel module intercepts this read system call and forwards it to the local RFS

server.

3. The RFS server iterates current member list. For each enabled member, RFS server finds out the corresponding real file descriptor, and conducts the read operation to that particular member. If the read operation succeeds on the current member, the RFS immediately return success to application. Otherwise, RFS will disable the current member and try the next available member.
4. If read on all members fail, RFS will return error to application.

d) File descriptor mapping

As we can seen from above, RFS must map a virtual RFS file descriptor to multiple real file descriptors, each of which represents an opened file on corresponding member. To do this, each RFS server maintain a mapping table from virtual file descriptor to real ones. The table is an array of link list. The virtual file ID will be the index of the array. Each array element is a link list, of which each member is a data structure containing the real file descriptor, the associated member.

A file descriptor mapping is added whenever a file is opened and removed when the file is closed.

e) Automatic file syncing algorithm

As members can join and leave dynamically, RFS also automatically syncs the data between them. For each member, RFS records the latest timestamp when it becomes enabled. So given a list of enabled member, the member with oldest enabled timestamp will have the most complete data. Therefore, whenever a new member joins the list, the system will sync the data from the oldest member to the new member. The data syncing logic will update the data on the new member if the file at destination doesn't exist or is older than the source.

f) Simple membership management

Currently the IP addresses of all possible members are stored at a file called rfs_config. In future we may support dynamic member discovery mechanism.

- Each member maintains a member client thread, which periodically (every 10 seconds) send JOIN message to all possible members. The JOIN message includes its IP address, local directory, and the time it becomes enabled. And when the local RFS server is shut down, a LEAVE message will be sent out.
- In the same time, each member contains a member server thread, which receives the JOIN or LEAVE message from other members.
    - ❖ Upon a JOIN message, the server thread will add or enable the member, update its timestamp, and mount the remote directory via NFS.

- ❖ Upon a LEAVE message, the server thread will disable the member.

5.2) Code

We have attached the programme code in Project.zip file.

6.0 Data Analysis and Discussion

- **Write Performance Analysis**

Table 1: Replicated write (backup) latency: Unit seconds

File Size	Three Members	Two Members	Single Member (local)
512KB	4.662	1.419	0.211
1MB	8.307	2.929	0.4
10MB	68.508	30.404	4.602

Table 2: Overhead of replicated write (backup) compared to single-member write

File Size	Three members	Two member
512KB	2109%	573%
1MB	1977%	632%
10MB	1389%	561%

Table 1 and Table 2 give the absolute and relative overhead of replicated write for multiple members with different file size.

Analysis:

1. The write latency increases with the increase in the number of members. Notably the increase of latency is not linear, i.e. faster than number of members. This is caused by
   - There is a machine in our experiment that shows very slow disk write and network transfer performance, so adding it as the third member greatly degraded the overall performance. This suggests that a single slow machine may become performance

bottleneck. And we should be cautious when choosing the machines for RFS.

- The network contention increases a lot after more and more members join in, because we need to sent more messages on the network.

2. The write latency increases linearly with the size of files. This result was expected. Since the overhead of replicated write was high, we implemented an improved version of replicated write called parallel writing – for every member write, we use a separate thread to perform write. So the latency will be slowest of all the members, instead of the sum of all members. The following Table 3 gives the improvement of parallel writing with three members.

Table 3 Improvement of Parallel writing compared sequential replicated write

File Size	Three members latency(s)	Improvement (%)
512KB	1.127	76%
1MB	2.209	73%

We can see the relative improvement of parallel writing is very significant, they are 76% and 73% respectively. This means multi-thread does help the write performance a lot as all heavy IO are performed concurrently

Read performance analysis

Normally we always read from local member, so it is very fast, the same as regular file read. But when local member disabled, we will do a fault-tolerant read from remote member. As we can see from Table 4, the remote read is much slower, which is mainly caused by network latency.

Table 4. Read performance from local and remote member

File Size	Read from local (s)	Read from remote (s)
512KB	0.035	0.091
1MB	0.052	0.160
10MB	0.27	1.152

7.0 Conclusion

Our Replication File System demonstrates the qualities essential for supporting small-scale data processing work-loads. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.
Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast recovery.

Our design improves the throughput of backup by concurrent writes and performance of data retrieval by read latency into account. It also makes the system cost and time efficient by implementing member management on independently on all nodes.

8.0 Future Work

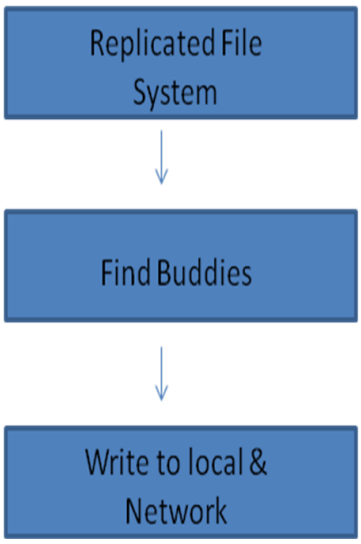
1. Data can be retrieved simultaneously from multiple machines. This would make the reads much faster.
2. More work can be done on the privacy front. The files can be encrypted and stored on other machines so that only the owner can access his files. Also, machines can be authenticated before they are allowed to join the network.
3. Scalability is another issue that requires more work.

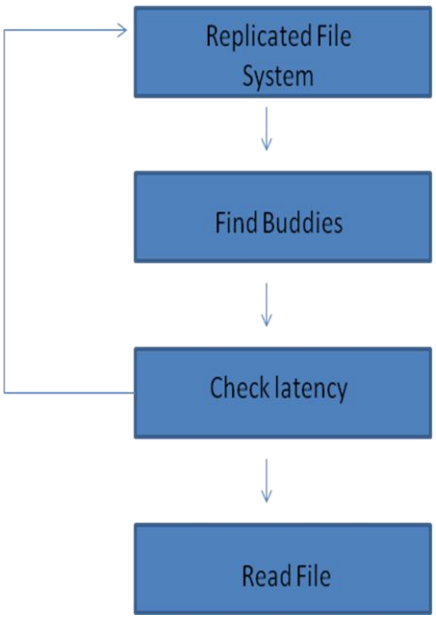
9.0 Bibliography

1. Ghemawat. S. and Gobioff. H. and Leung. S.T. 2003 The Google File System. http://labs.google.com/papers/gfs-sosp2003.pdf
2. Stender. J. 2009 Efficient Management of Consistent Backups in a Distributed File System
3. Cox. L.P. and Murray. C.D. and Noble B.D. 2003 Pastiche: Making Backup Cheap and Easy. http://www.usenix.org/events/osdi02/tech/full_papers/cox/cox.pdf
4. FUSE - http://fuse.sourceforge.net/
5. VMWare - http://www.ubuntu.com/getubuntu/download

10.0 Appendices

- **Program Flowchart**





- **Program Source Code and make file**

  rfs.c
  member.c
  common_header.h

header.c
Read.c
Makefile

- **Readme file**

RFS-Replicated File System

1. Install Linux virtual machine on windows using VMWare server 2.
2. Create user test/:test with the administrator privilege.
3. Install fuse-2.8.4.
4. Install NFS-Network File System.
5. Run: make rfs
6. Create rfs_config file,which includes IP addresses of all possible member machines, and put this file under the same directory as rfs executable files.
7. ./rfs –d eth0 $(real_path) $(virtual_path)

- **Other related material**

  - **RFS – Replicated File System built on FUSE**