A Fault Tolerant Fuse File System

Anusha Balaji and Taru Doodi
University of Florida

Abstract:

The paper implements Triple Modular Redundancy (TMR) to provide fault-tolerant network file system. Every operation done on a file is replicated on three different servers. The system is robust to faults in a single servers, i.e. when data gets corrupted or the server fails and restarts. It is also robust against a second server failing after the first one recovers. TMR is not robust against two server failures simultaneously. This paper evaluates a file system with TMR implemented and evaluates the performance of the fault tolerant fuse file system

Keywords: *Triple Modular Redundancy (TMR), Network File System, Fault Tolerance.*

Introduction

Fault tolerance is a system's ability to be resilient against faults and provide consistent performance. Redundancy is one of the most common approaches to make a system fault tolerant. This approach provides backups in situations where one piece of hardware becomes faulty. The redundant hardware can provide required functionality and extend transparent services to the user. Redundancy also finds application in load balancing when traffic on a particular device is high and needs to be equally distributed over the system. There are three techniques to implement redundancies: passive, active and hybrid. In passive technique the fault is hidden. An active method aims to detect, isolate and repair the fault. The hybrid method masking faults reduces the appearance of failures and the system is repaired by isolating faulty modules and repairing them.

TMR was first introduced by von Neumann[2]. TMR is a passive method for redundancy implementation and uses three servers and a majority vote is established for all the operations. The operation is termed successful only when two out of the three servers perform the operation (read and write) correctly. Therefore the TMR can detect and correct one error and fails for two errors. It can also be seen as that failure of one system in TMR is at fault but when 2 or more server failures occur the TMR module is at failure

The common cases of failure are when a server goes down or when the data on a server gets corrupted. The client cannot control the failure if two servers go down. In cases where a single server goes down, the client identifies the situation and returns the data if it matches that of the two active servers. The server that is down cannot be updated with a correct copy of data in such a case. Situations where data gets corrupted on one server, it is identified and rewritten with the correct data from the other two servers.

Background and Related Work

Network file systems enable data to be accessed in a networking platform. In such a system the data is stored on remote servers, and clients can accesses data by logging into the server. All the changes made to a file by the user are propagated and saved on the server. A single server for multiple clients can face issues such as excessive traffic and congestion. Hence, in such systems, scalability and reliability have become growing concerns. This has led to development of distributed file systems. The distributed file systems have high reliability and capable of handling much more traffic. Implementation of decentralized servers maintains multiple copies of the data. To make them fault tolerant, data replication [1] is spread across different servers within the same data centres and across data centres at different geographical locations. Big companies such as Google and Facebook maintain a minimum of three copies of user data in their data centres.

A lot of work has been done to optimize the various components of the TMR systems, such as the cost, improved voter design, etc. TMR finds applications not only in network file systems but also in regular hardware design. Redundant hardware is added to maintain the functionality in case one of the modules fail. A lot of work has been done with the aim to improve the majority voter design in hardware [5]. Applications including FPGAs also use TMR to implement the critical sections of the hardware design. To further address its importance, Xilinx provides a tool called *TMRtool* to make TMR implementation on FPGAs easier [3]. Work has been done to improve TMR response when two of the three servers fail by retaining its state [4].

Triple Modular Redundancy (TMR) Implementation

Fuse has been used widely to enable access for non-privileged users to access the file systems. A distributed fuse file system is a simple RAID 0 implementation of storing the files in a distributed fashion among the servers available. Corruption of data in any of the server or fault in any of the server acts as a single point failure for the entire distributed file system. In order to make the file system more reliable, fault tolerant techniques need to be applied on the file system to increase the robustness of the system. In this paper, fault tolerance in Fuse File System is incorporated using the Triple Modular Redundancy, where each copy of the server is replicated in two other servers. A majority voting is implemented to pick the correct data in case there is a corruption of one of the data.

Design

TMR is implemented in FUSE file system by making modifications to allocation of servers, storing the file on the servers, retrieving the file from the triplicated servers. When a particular file is corrupted or a server goes down, the file system uses the correct data from the remaining two copies, as TMR relies on majority voting, retrieves the correct value and also tries to restore the correct value on the server containing the corrupted value. The TMR implementation also uses a refresh procedure to do consistency check on the different servers. A refresh procedure checks for the data in the distributed servers on regular intervals of time. In this implementation, we have made the refresh interval to be a minute. Every minute, the servers are refreshed to check for consistency in their content. The project is an extension of the assignment 4 implementation, which has been modified to obtain the TMR design (modifications are discussed in detail below). The implementation also modifies the server code to expose refresh feature.

Server allocation and lookup: When the client starts three server bank objects, each with desired number of servers, unique URL for each server is created. Three servers, selected one from each server banks are allocated for every file that needs to be stored. Thus three copies of the data are stored for future access. Whenever a server has to be allocated for writing a new file, a new server from within the server bank is allocated using round robin allocation method. The round robin allocation is dependent on the number of files in the file system and not all the data present on the server. The index of the selected server is used to allocate a server of the same index in the other two server banks. This maintains consistency for server allocation. While reading a file the client looks for the server the file is stored in.

Reading data from server: When a file has to be read, a list of servers that contain those files is obtained and a ‘get’ is performed for that filename on the servers that contain the data. The data obtained from each of the servers is subjected to a majority voter check. Only when the majority is established the correct copy of data is returned to the file system. If no majority is established a key error is raised.

During this process the corrupted copy of the data is also identified. The corrupt copy is replaced by the correct data obtained from the other two servers (provided they are correct and form majority). Data restoration happens on all the three servers as it will take care of the metadata parameters such as last access time.

Writing data to server: When new file has to be written it will check whether that file exists in the file system. When it does not find an existing entry by that name, it will create a new entry for that filename

after allocating the servers. This is done by storing that key (refers to filename) corresponding to servers on which that file would be written. The data is then written to these returned list of servers It is ensured during server allocation that the list of server that is returned contains three servers, one from each of the server bank. An important reason to introduce server bank organization is to make it closer to the real world scenarios, where the copies are mostly stored in three geographically distributed datacentres.

Refreshing server: Refresh server is a very important feature in the design. It is essentially run on a separate thread which calls the function every 60 seconds. The refresh functionality is essentialto restore data into the servers that were down and did not get updated with the changes that happened in the other two actively running servers. The changes that could happen are deletion of a file, updating contents of the file, i.e. re-write or addition of a new file. To deal with these issues the refresh function checks the data present on all the servers, this includes comparing the contents as well as the metadata. This check is similar to a get function that is run for all existing files, i.e. keys on the server.

Since, our server implementation is relatively small this method of refresh works smoothly without creating any race conditions with other file operations. However, if the file system was large then the refresh itself would take up a lot of time and might run into errors while read and write functions happen. A proposed alternative solution is to use locks during the refresh operation and queue up all the read and write requests made by the user at the client end. When the lock is released after refresh the instructions in the queue will start to execute. For larger file systems, the intervals between refresh should be increased as we cannot have the client waiting frequently for long durations.

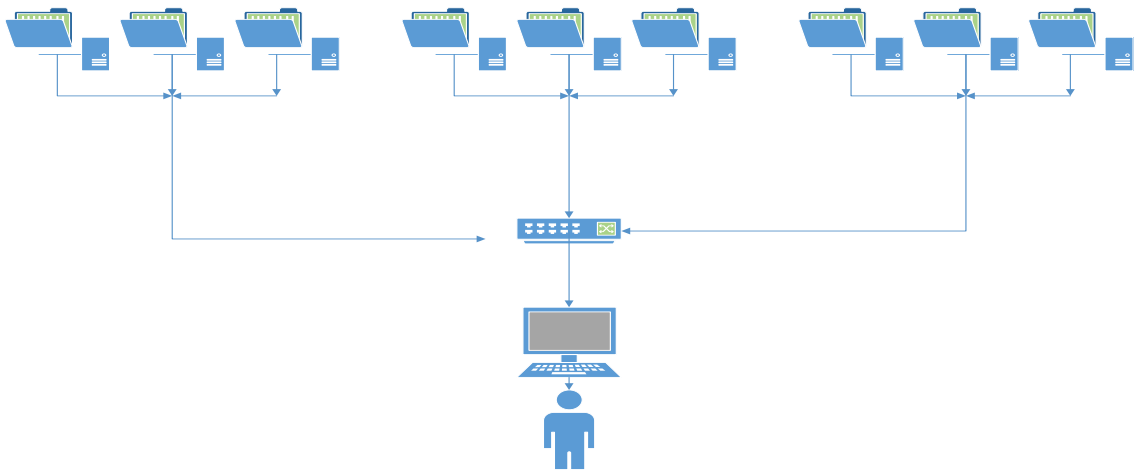

Figure 1 TMR implementation, with three servers in three server banks. The servers are accessible to the user through the majority voter.

The accesses to servers are limited by the files that are present in the file system. The user cannot access any other data introduced on the server through the file system. In this way, the server might be transparently used by other users for tasks which do not relate to the file system run by the user. In case a data on the server gets corrupted during an access by another user, the data is restored to original data during the refresh or get functions whichever happens first on the client side. This strict exclusion is derived from the fact that an error on any one server will not corrupt all three servers and will definitely not be able to influence metadata. If a malicious intent is made to sequentially corrupt the metadata and the data then the existing TMR would fail. However, if mutually exclusive operations are run on the server it can be used by more than one user at once. This functionality was exposed to us while devising a testing mechanism for the implementation.

Design Evaluation:

Test Cases:

The basic checks were performed to ensure the working of the TMR design as a reliable file system. To test the robustness of the file system, ServerProxy[6] objects of each running server were created and accessed via a separate terminal window. These objects were used to exploit all the functions exposed by the server such as the put, get and a function to print content (print_content). The overview of testing is covered here however the details of testing are covered in README file.

A) Consistency of Data writing:

When a data is written, the data is checked on all three servers using the print_content() on each of the server objects. This way one can also see the round robin assignment of servers when a file is being written.

B) Consistency during data reading:

The lookup for a particular file was done using its filename as the key. Before reading it compares the data and the metadata for the accessed filename. It is capable of detecting errors and those scenarios are discussed in detail in test case D.

C) Listing Metadata:

The name of all the files in the file system is obtained from the server that saves the root (‘/’) directory. Then the metadata for each of the listed files is obtained from the servers they are written on. The file system essentially fails completely when two server storing the root directory data cannot be accessed.

D) A cluster of servers goes down while execution or Servers down on start-up:

When running a distributed set of servers storing the file system, there is a good chance that one of the server might be down or might be interrupted while execution, making the file system fail for such cases. TMR implementation preserves the file system from going down, by leveraging the files on the replicated servers and thus maintaining the file system stable. The first case is to evaluate the robustness of the system when a server or a cluster of servers goes down. As long as a majority of the files is preserved in the system, the fault tolerant system manages to preserve the files and once the server comes back to functioning, the files get restored in the system. This case also evaluates when a server goes down while the file system is in operation. In both these cases, the file system yields the correct results.

The file system is equipped to refresh all the servers and check for consistency of data amongst the different servers. When a crashed server is brought back to its operational state or a spare server is brought back to replace the server, the data in the servers get refreshed and the contents get restored in the server.

The refresh function is spawned as a separate thread which checks for consistency amongst the server clusters periodically. The refresh function also checks if the server is functional. If it was brought back up within the 60s interval, the contents are restored.

Once the server gets refreshed and the contents restored, a failure on another server will be managed by the file system. As long as the Mean time between Failures (MTBF) is greater than the Mean Time To Repair (MTTR), the system gets restored to normalcy and can handle server failures.

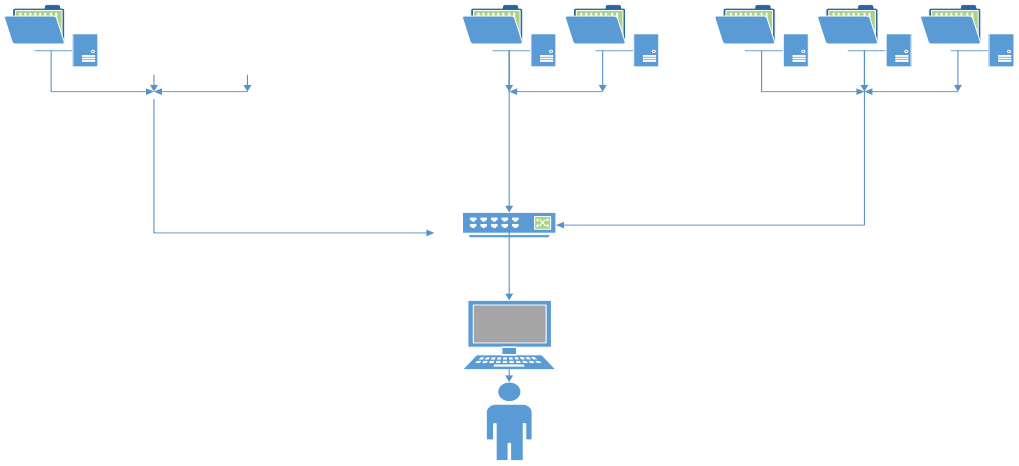

Figure 2 When one of the servers in a server bank is down

E) Corruption of data in one cluster of servers

This test case analyses the corruption of data in one of the servers. Any corruption in files in one of the servers is recovered when the file is accessed the next time. Repeated corruption of files is also recovered by the system. In case the corrupted file is not accessed in a long time, the file is automatically restored when the servers are refreshed at regular periodic intervals. This increases the reliability of the system when a file is corrupted and another server goes down provided the server fails after the the system is refreshed. This enhances the reliability of the entire system.

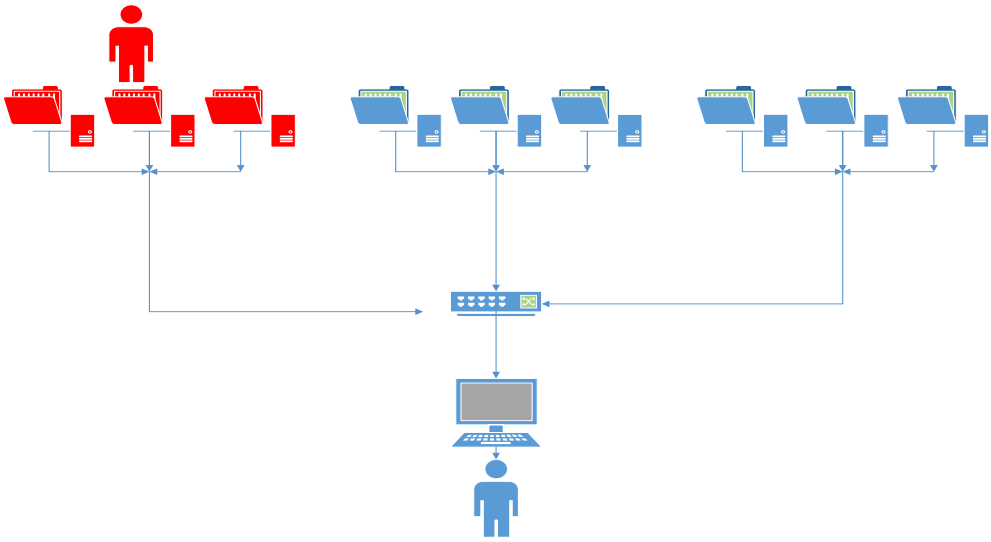

Figure 3 Corrupted data in the first set of Servers

F) Deleting keys not present on all the servers:

This scenario would occur when a server has been down for a while and a particular file that was on that server was deleted by the user. In such cases that file would be removed from two active servers but the server that could not be reached would not be aware of it. The refresh functionality in the system takes care of these scenarios. It periodically refreshes every minute and removes any additional keys that are not present on the majority of the servers. This would be helpful when a user may create a file with the same name at a later stage after deleting a previous copy. This also takes care of any extra keys entered by malicious entity and were not written by the user. The client identifies the keys and deletes them by setting the time to live as zero.

File System Benchmarking

Iozone [7] is a benchmarking tool useful for analysing the performance of various file systems. In this study, the fault tolerant fuse file system was tested using the iozone benchmarking tool to measure and contrast the performance of the triple modular redundant fuse file system with the normal fuse file system. The benchmarking tool helps to analyse the system performance such as sequential reads, writes, sequential re-reads and re-writes, random read, write, etc. It runs tests for different file sizes which are further broken down as smaller record sizes. A record size of a file is the smaller sections of file which are written at the same spot in the hardisk. Studying this is essential as this record size can contain spots in the disk which can fit within the TLB and hence increase the performance. Although this is not our major focus in implementation, the performance impact is evident as comparison between distributed and the redundant file system is made. Some of the important test definitions are covered as following:

Write: It measures performance of writing a new file. While creating a file the overhead incurred is higher as the file system has to take care of data storage in the media. This overhead is termed as "metadata". The metadata contains the information regarding various time stamps of the file, such as last access, modified, etc. In our case there is one extra access to the server where the root information is stored. This is necessary as the root directory must contain a list of all the files that are stored in the given directory.

Read: It measures performance of the file system while reading an existing file.

CPU Utilization: The CPU utilization is reported in percentage utilization of the CPU to perform that particular test.

The fault tolerant file system was benchmarked for different file sizes using the Iozone tool and the following results were obtained. We compare the performance of the TMR with respect to the original distributed file system.

Table 1 Comparison for Write Throughput (kBps)

File Size (kB)	TMR Servers				Distributed Servers			
	Record Sizes (kB)							
	4	8	16	32	4	8	16	32
4	138				701			
8	135	121			363	548		
16	98	125	145		322	399	442	
32	49	46	56	66	140	157	172	170

Table 1 contains the throughput values for write operations. This data supports the intuitive understanding that if the same work were to be performed thrice the performance would decrease by three. As the system bandwidth is now used three times more for a single job.

Table 2 Comparison for Read Throughput (kBps)

File Size (kB)	TMR Servers				Distributed Servers			
	Record Sizes (kB)							
	4	8	16	32	4	8	16	32
4	1421111				1310136			
8	1996610	2842223			1996610	2620273		
16	1433775	3225504	3993221		3225504	3993221	3993221	
32	358	302	5201089	7986442	1060	1006	5410762	5201089

Table 2 contains the data for read throughput in kiloBytes per second. The reduced read performances can be more clearly observed for higher file sizes (32kB) and lower record sizes (4 and 8 kB).

Table 3 Record Rewrite Throughput (kBps)

File Size (kB)	TMR Servers				Distributed Servers			
	Record Sizes (kB)							
	4	8	16	32	4	8	16	32
4	123				449			
8	102	101			255	308		
16	71	82	101		202	257	275	
32	39	50	59	62	125	147	158	165

Table 3 contains the data for rewriting records. Here we can again observe that the throughput approximately decreases by a third. However since the records are a part of the file and could be written to different locations, the read performance can vary very slightly due to that. This pattern is more apparent for a bigger file size(32kB).

Bugs encountered:

While running Iozone it was noticed that the current file system does not support files greater than 16kB sizes without running into key errors. This may be attributed to the high speed read and writes that the file system is subjected to by the Iozone benchmarks. We conclude this as the errors are not exactly reproducible and are somewhat erratic.

We should also take care while corrupting data for testing purposes. An extra pickling during corruption will not get removed and appear as extra string which would not be handled successfully by the fuse file system. It will constantly result into conflicted data.

Conclusion:

The implemented TMR successfully meets our set goal. It distributes and replicates data across the servers. It restores corrupt data during access as well as refresh. It also deletes files from a server when it comes back up. It also writes any new files that should have been written to it when it was down. The TMR implementation also allows server reuse/ sharing as it does not tamper with the data that does not belong to the file system. The functionality is tested with results for each of the listed cases, which are reproducible.

Work Division:

The project was completed with an amazing team spirit. The work was equally divided, well planned and completed within the set time frames as decided by us. The major part of coding was distributed as: Functions to refresh the server and majority voter (Anusha); Triplication of distributed file system and handling the metadata (Taru). However, the integration, analysis, testing and bug fixing were done jointly.

Reference:

[1] Peric, D.; Bocek, T.; Hecht, F.V.; Hausheer, D.; Stiller, B., "The Design and Evaluation of a Distributed Reliable File System," *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on* , vol., no., pp.348,353, 8-11 Dec. 2009.

[2] J. von Neumann, "Probabilistic logics," in Automata Studies. Princeton, N. J.: Princeton Univ. Press, 1956.

[3] http://www.xilinx.com/ise/optional_prod/tmrtool.htm

[4] Katsuyoshi Matsumoto, Minoru Uehara, Hideki Mori: "STATEFUL TMR FOR TRANSIENT FAULTS", In Proc. of the 8th International Symposium on Intelligent Automation and Control (ISIAC2010), No.511, pp.1-6, (2010.9.19-22, Kobe, Japan)

[5] W. J. Van Gils, “A Triple Modular Redundancy Technique Providing Multiple-Bit Error Protection Without Using Extra Redundancy,” *IEEE Transactions on Computers*, vol. C–35, no. 7, pp. 623–631, Jul. 1986.
[6] https://docs.python.org/2/library/xmlrpclib.html
[7] Iozone Documentation, www.iozone.org/docs/IOzone_msword_98.pdf

APPENDIX

The appendix contains the detailed results of Iozone benchmarking for

1. TMR servers
2. Distributed servers.

Iozone results for TMR servers

Command: iozone -a -+u -g 32k -n 4k -b ../tmr.xls

The top row is records sizes, the left column is file sizes

Writer Report				
	4	8	16	32
4	138			
8	135	121		
16	98	125	145	
32	49	46	56	66

Re-writer Report				
	4	8	16	32
4	106			
8	100	91		
16	73	84	90	
32	42	44	58	62

Reader Report				
	4	8	16	32
4	1421111			
8	1996610	2842223		
16	1433775	3225504	3993221	
32	358	302	5201089	7986442

Re-reader Report				
	4	8	16	32
4	1905268			
8	1996610	2620273		
16	3077584	3225504	3225504	
32	320	322	5410762	6451008

Random Read Report				
	4	8	16	32
4	126			
8	207	158		
16	274	274	276	
32	40	107	325	346

Random Write Report				
	4	8	16	32
4	145			
8	95	107		
16	76	86	94	
32	38	50	60	63

Backward Read Report				
	4	8	16	32
4	83			
8	77	101		
16	57	106	178	
32	38	82	135	239

Record Rewrite Report				
	4	8	16	32
4	123			
8	102	101		
16	71	82	101	
32	39	50	59	62

Stride Read Report				
	4	8	16	32
4	123			
8	202	143		
16	286	259	256	
32	295	299	325	340

Fwrite Report				
	4	8	16	32
4	128			
8	145	91		
16	96	185	95	
32	42	66	123	62

Re-fwrite Report				
	4	8	16	32
4	128			
8	161	104		
16	96	162	91	
32	43	67	122	62

Fread Report				
	4	8	16	32
4	107			
8	168	177		
16	265	253	260	
32	131	168	329	346

Re-fread Report				
	4	8	16	32
4	120			
8	152	166		
16	288	253	264	
32	82	177	327	329

(Zero values should be ignored)

Writer CPU utilization report				
	4	8	16	32
4	3.642195			
8	0	0		
16	0	0	0	
32	0.095557	0.109359	0.430585	0

Re-writer CPU utilization report				
	4	8	16	32
4	0			
8	2.722202	0		
16	0	0	0	
32	0.090239	0.102929	0	0

Reader CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	3.291816	0	0	
32	0.26087	0	0	0

Re-reader CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0

Random Read CPU utilization report				
	4	8	16	32
4	0			
8	0	4.061556		
16	0	0	0	
32	0.113594	0.356034	0	0

Random Write CPU utilization report				
	4	8	16	32
4	7.993873			
8	0	0		
16	1.214601	0	0	
32	0.084692	0	0.51617	0.439189

Backward Read CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	2.2892	0	
32	0.078531	0.32811	0	0

Record Rewrite CPU utilization report				
	4	8	16	32
4	0			
8	2.91369	0		
16	0	0	2.198414	
32	0	0.282585	0	0

Stride Read CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0

Fwrite CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	1.328267	0	0	
32	0.249528	0.296051	0	0

Re-fwrite CPU utilization report				
	4	8	16	32
4	0			
8	0	2.638032		
16	0	0	0	
32	0	0	0.412345	0.545523

Fread CPU utilization report				
	4	8	16	32
4	3.69943			
8	0	0		
16	0	4.446211	0	
32	0.493633	0	0	0

Re-fread CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	3.886118	
32	0	1.035202	0	0

Iozone results for Distributed servers

Command:iozone -a -+u -g 32k -n 4k -b ../dist.xls

The top row is records sizes, the left column is file sizes

(Zero values should be ignored)

Writer Report				
	4	8	16	32
4	701			
8	363	548		
16	322	399	442	
32	140	157	172	170

Writer CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0.930038	0	0	0

Re-writer Report				
	4	8	16	32
4	342			
8	281	379		
16	191	253	264	
32	122	151	154	174

Re-writer CPU utilization report				
	4	8	16	32
4	0			
8	8.106445	0		
16	0	4.31023	0	
32	0	0	0	0

Reader Report				
	4	8	16	32
4	1310136			
8	1996610	2620273		
16	3225504	3993221	3993221	
32	1060	1006	5410762	5201089

Reader CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0

Re-reader Report				
	4	8	16	32
4	1421111			
8	673706	4298711		
16	3225504	3077584	5240547	
32	1010	1020	1101900	7986442

Re-reader CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0

Random Read Report				
	4	8	16	32
4	373			
8	627	551		
16	753	772	786	
32	133	330	856	962

Random Read CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	7.124685	
32	1.052639	0	0	0

Random Write Report				
	4	8	16	32
4	343			
8	258	329		
16	215	219	274	
32	122	142	157	170

Random Write CPU utilization report				
	4	8	16	32
4	8.74301			
8	0	0		
16	3.406976	0	0	
32	0	1.64456	2.330814	0

Backward Read Report				
	4	8	16	32
4	264			
8	227	383		
16	175	320	523	
32	130	232	399	600

Backward Read CPU utilization report				
	4	8	16	32
4	0			
8	7.524765	0		
16	0	0	0	
32	0	0	0	0

Record Rewrite Report				
	4	8	16	32
4	449			
8	255	308		
16	202	257	275	
32	125	147	158	165

Record Rewrite CPU utilization report				
	4	8	16	32
4	0			
8	0	9.990298		
16	0	3.075957	3.556187	
32	1.269855	0	0	1.975758

Stride Read Report				
	4	8	16	32
4	318			
8	571	496		
16	767	820	604	
32	1002	1069	897	869

Stride Read CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0

Fwrite Report				
	4	8	16	32
4	505			
8	498	379		
16	264	499	258	
32	140	198	326	163

Fwrite CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	2.225594	0	0	
32	0	1.637675	1.696113	0

Re-fwrite Report				
	4	8	16	32
4	429			
8	533	334		
16	283	493	288	
32	139	206	305	173

Re-fwrite CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	3.7452	0	
32	1.419333	0	0	0

Fread Report				
	4	8	16	32
4	279			
8	673	488		
16	751	858	881	
32	513	476	878	931

Fread CPU utilization report				
	4	8	16	32
4	15.15172			
8	0	0		
16	0	0	9.545352	
32	0	0	0	0

Re-fread Report				
	4	8	16	32
4	423			
8	575	518		
16	768	814	733	
32	525	520	909	967

Re-fread CPU utilization report				
	4	8	16	32
4	0			
8	0	0		
16	0	0	0	
32	0	0	0	0