UNIVERSITETET I OSLO
Institutt for informatikk

Comparison of NFS, Samba and OpenAFS

Masteroppgave

Claudia Eriksen

19th June 2005

COMPARISON OF NFS, SAMBA AND AFS

By
Claudia Eriksen

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
OSLO UNIVERSITY COLLEGE
JUNE 2005

Abstract

By making use of a distributed file system, users of physically distributed computers are allowed to share data and storage resources. This paper compares three distributed filesystems: the Sun NFS filesystem, Samba and OpenAFS. The emphasis of the comparison is on the functionality, management and performance of these three file systems in a heterogenous network. The iozone filesystem benchmark tool was used to gathered information about how well these filesystems performs read/write operations, in order to numerically characterize the performance. Analyzing and properly understanding the data gathered from the measurements, reveals that Samba and NFS perform almost equally on windows machines equipped with different Operating Systems and hardware. On a client running SuSE9.1 NFS has the best performance for both read and write operations. OpenAFS had the lowest performance on windows machines and on the client running it performed as well as Samba. The paper concludes with a recommendation for which distributed filesystem best suits which computing environment.

Acknowledgements

First and foremost I thank my advisor, Simen Hagen, for his help in seeing this job through and for not giving me up when I was not far from giving up.

I thank my teacher, Mark Burgess - the best teacher that I haver known - for his tremendous sacrifice, help and support during the last three years of my school education.

My husband, Ole Petter, deservers greater thanks than I can possibly give. For more than 6 years he has patiently allowed me to pursue my dream, never complaining it was taking too long. On days when I was doubted everything I'd done and thought I could do no more, he comforted me and gave me the strength to keep going. Without he there for me I surely would have quit. Thank you so much, my dear. I love you.

Contents

Chapter 1

Introduction

Distributed systems keep growing in scale and importance and thus sharing of data in these systems is getting more and more pervasive. Sometimes users are forced to physically move around in a network and being able to access their data from anywhere is important. The ease of data sharing enhances the value of distributed systems to its users.

This paper compares three distributed file systems under different Operating Systems. NFS (Network File System) was first introduced in 1985 by Sun Microsystems. It is the most commercially successful and widely available remote filesystem protocol[29] today. It uses Remote Procedure Call (RPC) method for communication between computers. AFS (Andrew File System) is a file system originally developed at CMU (Carnegie Mellon University) , and developed as a commercial product by the Transarc Corporation, which in 1998 become fully-owned subsidiary of IBM. IBM made a copy of the source available for the development and maintenance community, a release they called OpenAFS. Samba, an open source project from the Samba Team that extends CIFS protocol, allows Windows clients to connect to servers running Linux and other operating systems for file-sharing, printing, and user authentication.

These three distributed filesystems share enough features that a comparison is interesting. Even though there are differences in their implementation, they all have the same concept. They provide access and location transparency to the end users. By location and access transparency is meant that users regardless of which machine on the network they log on, have the ability to access and

view their files in exactly the same way.

The emphasis of the comparison is on the functionality, management and performance in a heterogenous network. The IOzone filesystem benchmark tool was used to gathered information about how well they perform read/write operations, in order to numerically characterize their performance. The measurements were made on 7 computers running different OS and equipped with different hardware. Three of the workstations worked as file servers, and the rest served as clients. One of the clients was a linux workstation and the other three were windows machines. The performance results are presented in two groups.

The first group parallels the analysis of each distributed filesystem on each client. The bottom lines are that NFS's average read/write performance is a 100% higher than its average read/write performance on windows clients. OpenAFS had a better average write performance on the *linux client* than on windows clients. Its average read performance was approximately the same on all clients. Samba's average write performance was a bit higher on windows clients than on *linux client*, for file sizes under 16 Mbytes. beyond this edge the average write performance was comparably on all clients. Its average read performance was much higher on one of the windows clients and comparably on the other clients.

The second group compares the performance of the distributed filesystem with each other. The bottom lines are that NFS's read/write performance, on linux (SuSE9.1) is considerably higher than both OpenAFS and Samba which performs as well as each other with small variations. On windows clients, NFS and Samba had similar performance for both read and write operations, and performance that is much higher than OpenAFS's performance.

1.1 Motivation

As local networks of workstations are in continuously growth, the need to share resources, and the management of users and system files becomes increasingly more important. The most used distributed filesystems, as of today, are NFS, Samba and OpenAFS. Suppose you want to migrate from

one filesystem to another or your computer environment has grown to a size that you consider to start using one of these filesystems. You would like to have more than just the source code and an administrative overview to go on before deciding whether to spend time learning and a filesystem or not. Since there are no available studies that shows how these three filesystems performs compared to each other, we would like to mape such a performance comparisons.

1.2 Research goals

The paper's goal is to create a comparative environment to gather enough read/write performance statistics based on which a comparison can be drawn outlining which filesystem performs best under the given conditions (hardware and operating system).

1.3 Thesis structure

The basic concepts underlying the design of each studied filesystem and their solutions to different issues that a distributed filesystem may encounter is presented in chapter 2. Chapter 3 describes the performance testing environment. Chapter 4 and 5 presents the write and read performance results respectively. The reader is introduced to the concept of measurement uncertainty explaining why the results can not be seen on as being accurately in chapter 6. Conclusions and further work are presented in chapter 7 respectively chapter 8.

1.4 Related work

There is very little information available about performance comparisons of NFS, Samba and OpenAFS. The author is not aware of any kind of studies related to this topic.

Chapter 2

Distributed file systems

This section describes the distributed filesystem concepts and should give the reader the necessary background information to better understand how OpenAFS, NFS and Samba implement these concepts. In the last part of the first section, the reader is briefly introduce to the Kerberos protocol. This is due to OpenAFS using a similar conception.

A distributed filesystem (DFS) provides a framework in which access to files is permitted regardless of their location. According to the way the file storage is managed, there are two DFS concepts:

- *server-client*: in this model a set of machines, known as servers (*a computer or device that manages the network resources*), provide storage for all of the files in the DFS. All other machines, known as clients (*PCs on which users run applications*), must direct their file references to these machines. Servers often run on dedicated machines enabling clients to be more general and often simpler to install and support.
- *peer-to-peer*: in this model each machine provides storage on its own attached disk, and allows others to access it remotely. A participating machine may act as both a client and a server.

2.1 Server/client environment

In an server/client environment, clients and servers do not necessarily have to be different machines (servers may also be clients of the services they provide). *File severs* are servers that provide access to centralized data. When a server delivers data to a client, it has to keep a record that it has done so in order to prevent a second client, that retrieved the same information, from editing the data, if another client has already modified the data. *Consistency* is known as the process by which the distributed filesystem guarantee that the copies of the data located on client machines are the same as the data on the file server. Consistency may be achieved by locking the file on the server whenever a client requests it. This way of guaranteeing consistency may be seen on as being quite crude. If,at the same time, the client that requested the data, only wants to look at the file, while another client actually wants to edit the data, will lead to a total waste of time for the second client. Distributed filesystems often makes use of a mechanism called a *callback* to provide a solution to this kind of problem. A callback is the process by which a server notifies a client whenever changes are made to the file that the client has retrieved form the server, sending a message to the client that it has to re-retrieve the file.

2.1.1 Transparent access to data

Making centralized data available to users without requiring the users to know whether their files are local or located on a remote file server is referred to as *transparent access to data*. In other words, regardless of on which machine in the computing environment the users will log in, they will see the same view of the distributed filesystem.

2.1.2 Namespaces

Namespaces are all the lists and directories in a computer environment (e.g. user accounts, available printers, names of hosts, Ethernet addresses). Every namespace has attributes for each element (e.g. user accounts have UID, home directories,owners, and so on).

2.1.3 Caching

When a client requests a file from the server, it has to have somewhere to put it, usually a portion of the disk known as a *cache*. A request first passes through the cache, and if the data is already there, the application requesting the file gets a pointer to the data in the cache without requiring any network traffic. Hence, the speed and efficiency of accessing files is improved, the network traffic is reduced and the server may operate more efficiently. Keeping a copy of the file in a cache on a client's local disk is known as *caching* that file information. Preserving and reusing cached information when the system is restarted is known as *persistent cache*. Thinking that there is a big possibility that the users of a client would want to work with the same files they worked with before the system went down, and that the files are still in the cache and haven't changed on the file server, persistent cache helps improving the performance.

Also, the network traffic can be reduced by keeping recently accessed disk blocks in a cache, since new accesses request to the same data block is handled locally. Thus how effectively data caching is performed, is an important factor in file system performance and scalability.

Data can be cached either on disk or in main-memory. Disk caches are more reliable and the cached data kept on disk are still there during recovery and don't need to be fetched again. Main-memory caches allow workstations to be diskless and data can be more quickly accessed.

Read-ahead caching: a larger block of data, than the requested data, is loaded into the cache when satisfying a request. Hopefully the next request can be served with the data already in the cache.

Write-behind caching: small writes are sent delayed to servers in the hope that the client will shortly ask for a new small write, and only a single message needs to be sent to the server.

With the use of cache, the *cache consistency* (changes made by a client to a file must be made available to another client when the file is read) is introduced.

Filesystem layout should optimize for disk writes since large caches may avert most disk reads [14].

2.1.4 Replication

The capability of distributed filesystem to provide online copies (replicas) of existing portions of the distributed filesystem is known as *replication*. When a server fails, replicas are made instantly and transparently available to users without interrupting their work.

2.1.5 Crash recovery

In a file server environment, if the server goes down, all the application servers accessing the shared storage are essentially down. If their critical data is on the shared storage, there is nothing that can be done until the file server is back on line.

A stateful server is a server that is able to maintain some information about its clients between servicing their requests. In opposition, when the server does not maintain any information on a client, we say the server is stateless.

Typically, when clients request a file from a stateful server a connection (session) between server and client is established. Before any access to the file is given to the client, the server fetches information about the file, stores it in its memory and sends the client a connection identifier. This identifier is unique to the the client and the file. It is used by the client during all the subsequent accessing until the session is closed. When the session is closed, the server reclaims the memory used by its clients that are no longer active.

The difference between a stateful and a stateless server is evident when a crash occurs during a service activity. When a stateful server is back on-line after such a crash it needs to restore the session state. In the case of client failure, the server needs to be notified in order to reclaim the memory space allocated to record the state of the client. When a stateless server crashes, its clients keep retransmitting their requests until they get a response.

2.1.6 Kerberos

Kerberos is network authentication protocol by MIT as a solution to network security problems. With username/password authentication methods the user is authenticated to each network service. Kerberos on the other hand, uses a trusted third party, a so called Key Distribution Center (KDC), to authenticate users to a group of network services. The protocol uses strong cryptography, based on secret-keys, so that a client can prove its identity to a server and vice-versa (concept called *mutual authentication*) across an insecure network connection.

How Kerberos works

When a user starts the authentication process, its principal (a unique identity like username) is sent to the KDC in a request for a Ticket-granting Ticket (TGT) from the Authentication Server (AS). If the AS knows about the principal existence, it will generate a random session key for use between the user and the KDC and a ticket encrypted with a session key known only to the SA and the KDC. The KDC encrypts this ticket with the user's private key and returns it to the user. The private key is derived from the user's password and is only known to the user and Kerberos. Now that the user has received the response, he/she is prompted for its password which is converted to a key that can decrypt the message sent by Kerberos and the authentication process is terminated. The TGT usually expires after just a few hours so that a compromised TGT is of use to an attacker only for a short period of time.

Note: *The standard AFS clients that perform authentication discard the TGT after they acquire an AFS service ticket. This means that OpenAFS users can't get tickets for other services using their AFS token. OpenAFS can be set up to work with regular Kerberos instead of AFS Kerberos.*

2.1.7 Time synchronization

Why would time synchronization between servers and clients be important? Suppose you as a user would like to update a remote file, but you get an error message saying that the version of the file on the file server is newer than your version. But you know that isn't true. Wouldn't that be irritating?

OpenAFS makes use of tickets called tokens that have an default expiration time of 25 hours. Suppose you have just authenticated yourself for the server, but half an hour later your tokens expire. You have to re-authenticate yourself. Or you can't even authenticate because your machines clock is behind the servers clock.

NFS does not synchronize time between client and server. In NFS environment the client has no mechanism for determining what time the server thinks it is. The consequence (if drifting clocks) is that a client may update a file, and have the timestamp on the file be either some time in the future or in the past, from its point of view. With NFSv3 a client a client can specify the time when updating the file, but the mechanism isn't widely implemented [23]. And it does not help when two clients, which clocks differs, are accessing the same file.

Samba offers a functionality called *time server*, a server that tells its clients what time it is. OpenAFS offers its own NTP.

NOTE: *when installing a file server, you should think about synchronizing the clocks between the machines participating in your network. Network Time Protocol (NTP) can be used.*

2.2 Sun NFS

The Network File System is a protocol in which a client can send one request and receive multiple data packets in return from a server. It is based on Sun's RPC version 2 protocol. NFSv2, published in 1985, was a 32-bit implementation of the protocol and used UDP exclusively as its transport mechanism. NFSv3, published in 1994, added TCP to its transport mechanism and was extended to 64-bit files.The latest version, NFSv4, published in 2000, is well-suited for complex WAN deployment and fire-walled architectures, has a stronger security (public and private key) and improved multi-platform support. It adds persistent, client-side caching and support for ACLs.

An overview of NFS is presented in [29]. Details of its design and implementation are presented in [6], and comments of NFS portability are presented in [5].

2.2.1 Naming and location

Naming is a mapping between physical and logical objects. Users work with file names representing logical data objects, while the system deals with physical objects (blocks of data) stored on disk.

NFS makes no distinction between clients and servers. That is, a workstation may behave as a server, exporting files, and may also behave as a client, requesting file access on another workstation. Configuring your network such that a small number of nodes run as dedicated servers, while the others run as clients, is a good practice for installation.

Each NFS client sees a Unix file namespace with a private root. The subtrees that the NFS servers export are bound to this root file system, by using an extension of the Unix mount mechanism. Since the mounted subtree may be renamed on the client side, there is no guarantee that the shared *namespace* is identical at all workstations. Only previously mounted remote directories can be accessed *transparently*.

2.2.2 Caching and replication

NFS clients caches disk blocks in the main memory (I/O buffer cache). Therefore, even if present, local disks are not used for caching. NFS uses a data caching scheme that relies on *polling by the client*. At the same time as a client caches a file, a timestamp is cached. This timestamp indicates the time when the file was last modified. This is used in validating the data before it is cached, always performed when a file is opened. When a file is opened, a cache validation check is performed on its parent directory as well. If the cached timestamp coincides with the timestamp on the server, the client pulls the data. If they don't coincide, and the server's timestamp is more recent, the client machine invalidates the cached data and reattaches them on demand. When data is placed into the cache, it is considered valid for a short length

of time. During this time period the client will use the cached data without verifying its modification time with the server. Directory caching for reading is performed in a similar way as file caching, but modification to them are performed directly on the server. File and directories don't have the same revalidation intervals. The techniques used between the server and the client may be either read-ahead or delayed-write[25].

To shorten windows of time in which inconsistent data could return from the cache, NFS uses a *flush-on-close policy*. When a file is closed, all partially filled NFS buffers are written to the NFS server. The problem is when another clint opens the file before the first one has closed it. To prevent this from happening the file locking mechanism is used.

Replication is supported by newer versions of NFS through a mechanism called *Automounter*[23, 31]. By using Automounter mount points may be specified as a set of file servers rather than a single server. Automounter will then make use of the most responsive server to first use. That is the server that first respond when a client requests a mount point. A set of volumes (a volume can be one or more subdirectories of an exported filesystem) may also be defined to use in series: when/if one volume fails to mount, Automount tries the next one in the list until one succeeds.

2.2.3 Crash recovery

NFS is a stateless protocol. Hence, the server does not need to keep information about which clients it is serving or which files the clients have open.

A benefit of this approach is that there is no need to do state recovery after a server or client has crashed and rebooted, making the NFS crash recovery simple and normally transparent to the user program. When a server crashes, the client re-sends the requests until a response is received (data will never be lost due to a server crash) and the server does no crash recovery at all. Since file operation requests are retransmitted several times without getting any response, these operations should and are *idempotent*. An idempotent operation has the same effect and returns the same output if executed consecutively[3].

Whenever an NFS client opens a file it will receive *file handles* (data structures that identifies the association between servers and their files) from a server. In this way an NFS server that becomes available after a crash, will be able to recognize which file the modified data belongs to. When a client crashes no recovery is necessary for either the server or the client.

2.2.4 Security

The mechanism used by NFS when performing access checks is based on the underlying Unix file protection. Each RPC request from a client sends the user's identity along with the request. The server assumes the identity, and each file access while servicing the request is handled as if the user had logged in directly to the server.

In the earlier versions of NFS, mutual trust was assumed between all participating machines. The client machine determined the user's identity which was accepted, without further validation, by a server. Requests made on behalf of *root* were treated by the server as if they come from a non-existent user, *nobody*. Hence, root received the lowest privileges for remote files.

With more recent version of NFS higher level of security was introduced. To validate RPC requests, DES-based mutual authentication was used. The common DES key needed for mutual authentication is obtained from information stored in a publicly readable database which stores a pair of keys, suitable for public key encryption, for each user and server. One key of the pair is stored in clear, is stored encrypted with the login passwd of the user. Any two entities registered in the database can deduce a unique DES key for mutual authentication. The mechanism is described in [7, 8].

2.2.5 System management

The system management in NFS may be achieved either with *Yellow Pages* (YP) or *Automounter*. If one does not know exactly what one is looking for, but needs a list of possible categories to match, such as in browsing for users or services, then the service is referred to as Yellow Pages. A number of UNIX database such as those mapping usernames to passwords, hostnames to

network addresses, and network services to Internet port numbers are stored in YP. YP provides a shared repository for system information that changes relatively infrequently and that does not require simultaneous updates at all replication sites.

Automounter is another mechanism simplifying system management. It allows a client to evaluate NFS mount points, thus avoiding the need to mount all remote files of interest when the client is initialized. Automounter can be used in conjunction with YP to substantially simplify the administrative overheads of server reconfiguration.

2.3 OpenAFS

OpenAFS is an open source implementation of the Andrew file system, a *distributed filesystem that enables cooperating hosts (clients and servers) to efficiently share filesystem resources across both local area and wide area networks*[1].

Some distributed filesystems provide consistency at a cost to performance. Other distributed filesystems provide good performance, but with weak consistency guarantees. The Andrew filesystem attempts to provide good performance as well as consistency guarantess[19].

A high-level overview of AFS is described in [12]. Details of its architecture and design are described in [19, 20, 17, 18].

2.3.1 Naming and location

The Andrew file system's namespace consists of a *shared* and a *local* name space. The shared name space is identical on all workstations, thus providing location transparent. The local name space contains only temporary files or files needed for workstation initialization and is unique on each workstation. The shared name space hosts users files giving users a consistent image when they move from one workstation to another.

As in UNIX, both name spaces are hierarchically structured. The shared name space is divided into subtrees, and each of such subtree is assigned to a single server, called its *custodian*. Each server maintains a copy of a fully replicated location database that maps files to custodians.

2.3.2 Caching and replication

Files in the shared name space are cached in fragments on the local disk at the client and by using callbacks cache consistency is maintained. A cache manager runs on each workstation. The cache manager maintains information about the identities of the users logged into the machine, finds and requests data on their behalf, keeps copies of accessed files on local disk, and uses the ordinary UNIX block cache to keep frequently used data blocks in main memory[17]. When a file is opened, the cache manager checks the cache for the presence of a valid copy. If such a copy exist, the open request is treated as a local file open. Otherwise an up-to-date copy is fetched from the custodian. Read and write operations on an open file are directed to the cached data. If a cached data is modified, it is copied back to the custodian when the file is closed.

Rather than requiring periodic verification of a files consistency as NFS does, AFS reduces server load by using callbacks: the server maintains a list of all cached copies of each data file and notifies clients when another client modifies the file. The client fetches the new data from the server the next time it opens the file. AFS clients send all modified data to the server when a file is closed, guaranteeing that the server has the most current version of the data and allowing the server to know when to invalidate the other cached copies. The use of callbacks, rather than checking with the custodian on each open, substantially reduces client-server interactions.

Clients also cache directory information in write through directory caches. All modifications to directories are sent immediately to the server, which maintains callbacks to keep cached copies of directory information consistent.

Read-only replication of data that is frequently read but rarely modified is

done to enhance availability. Subtrees that contain such data may have read-only replicas at multiple servers, thus the server load is evenly distributed.

Lock and unlock operations on files are performed directly on its custodian. If a client does not release a lock within 30 min., it is timed out by the server.

2.3.3 Crash recovery

The AFS cache manager will fail over to another server if the primary server fails. Client cleanup is addressed by the time-outs on the server callback registration. The server can delay service after booting for this period of time and know that clients do not expect any call backs. If the server were to crash during a write operation, the client would get an error.

Obviously, an AFS server is not stateless. An AFS server is forced to keep track of its outstanding promises so that callbacks can be revoked when necessary. To avoid becoming overwhelmed by callback management, an AFS server attaches an expiry to each callback promise. Furthermore, an AFS server is free to revoke callback promises with abandon, should the overhead of maintaining a large amount of state information become overwhelming. Crash recovery is complicated by the AFS server's cache invalidation mechanisms. When an AFS server restarts, it has no record of the callback promises it made before the crash.

If a fileserver crashes, the client's locally cached file copies remain readable but updates to cached files fail while the server is down.

Also, if the AFS configuration has included replicated read-only volumes then alternate fileservers can satisfy requests for files from those volumes.

2.3.4 Security

The design of Andrew file system attaches importance to security. The servers do not trust neither the network nor the workstations.

AFS uses an access list mechanism (applied to folders rather than files) for protection. The total rights specified for a user are the union of all the rights

collectively specified for him and for all the groups he belongs to. An access list can specify negative rights: denial of given rights. Negative rights are intended primarily as a way of rapidly and selectively revoking access to critical files and directories.

AFS establishes user's identity through a trusted third party by using something similar to Kerberos authentication (the kdc server is called kaserver). In the process, passwords do not travel over the network. During the process the user have to prove that he/she knows the password by being able to decrypt encrypted messages with the key derived from the password. Mutual authentication is ensured by a Kerberos server ticket: The user presenting the key can safely be assumed by the file server to be a legitimate user, and the user can safely assume that the server is genuine. A ticket with a limited lifetime accompanies each session and time stamps based on randomly generated session key are used by the protocol. Thus replay attacks are very difficult.

NFS operates "on top of" local filesystems; in other words it moves the data between machines but has nothing to do with how it is stored on the fileserver. On the other hand, AFS volumes are specially formatted for AFS, and are accessible only through AFS, even locally. Thus an organization deploying AFS has to trust it from the beginning, and has to make an instantaneous transition from no AFS to all AFS, referring to access to the volumes being converted.

2.3.5 System management

The operational mechanisms of AFS are built around a data structuring primitive called a *volume*[21]. A set of files forming a partial subtree of the server name space defines a volume. There is usually one volume per user and disk storage quotas are applied on a per-volume basis. A read-only replica of a volume can be created by a clone operation. Such replicas can be used to improve availability and performance.

Volumes also form the basis of the backup and the restoration mechanism. To backup a volume a frozen snapshots of its files is created by cloning. An asynchronous mechanism then transfers this clone to a staging machine from

where it is dumped to tape. To handle the common case of accidental deletion by users, the cloned backup volume of each user's files is made available as a read-only subtree.

A standardized set of protocols and naming convention[9] that a cooperative group of *cells* adhere to, provide the image of a single name space. This mechanism addresses key issues like cross-cell authentication and translation of users identities in different administrative domains.

2.4 Samba

Samba is a suite of Unix applications that speak the SMB (Server Message Block) protocol. It is an open source CIFS Server implementation which has grown to include tools, utilities and even ftp like clients started in 1991 by Dr. Andrew Tridgell. It consists of two key programs: smbd and nmbd. Their job is to implement the four basic modern-day CIFS services, which are: file & print services, authentication and authorization, name resolution and service announcement also called browsing. File and print services are, of course, the cornerstone of the CIFS suite. These are provided by smbd, the SMB Daemon.

Microsoft DFS technology has been implemented in Samba. It allows data to be accessed from a single share and to be distributed across multiple servers. The package has to be configured with the *–with-msdfs* option. The Windows clients need no additional software to interface with.

2.4.1 Naming and location

The name resolution and browsing, are handled by nmbd. These two services basically involve the management and distribution of lists of NetBIOS names.

Name resolution takes two forms: broadcast and point-to-point. A machine may use either or both of these methods, depending upon its configuration. Broadcast resolution is the closest to the original NetBIOS mechanism. Basically, a client looking for a given service will broadcast the request and wait

for a response. This can generate a bit of broadcast traffic (a lot of shouting in the streets), but it is restricted to the local LAN so it doesn't cause too much trouble.

CIFS, thus Samba, supports mounting multiple servers and disk volumes to subtrees in clients directory hierarchy to appear as if residing on the same server and volume. This is done in a very similar fashion with the way AFS mount remote servers. Changes in the physical location of the data caused by server reconfiguration are made transparent to the client (as long as their names remain consistent).

2.4.2 Caching and replication

Samba supports both read-ahead and write behind caches. Applications can register with the server to be notified whenever a file or directory is changed. Such updates help to avoid the problem of clients having to constantly poll the server in case they need consistent information.

File locking in samba may be controlled through several smb.conf parameters:

- blocking locks: when a client repeatedly requests a file lock, the server will notify the client when it becomes possible to grant the requested lock.
- oplocks: enables the client to cache file accesses, thus network performance can be improved.
- kernel oplocks: enables server side programs to break a Samba oplock. The result is improved data consistency when files are accessed both via Samba and via local processes or an NFS server. Unfortunately, not all Linux-like systems support this feature; only Linux 2.4.x and later kernels and IRIX.
- level2 oplocks: enables Samba to downgrade an oplock from read/write to read-only status when a second client accesses a file, rather then revoking an oplock entirely. Thus, improved performance.

By using *rsync* (replacement for rcp that has many more features)[22] data can be replicated across the network. This works bet for read-only data, but

with careful planning can be implemented so that modified files get replicated back to the origin point.

rsync uses the "rsync algorithm" which provides a very fast method for bringing remote files into sync. It does this by sending just the differences in the files across the link, without requiring that both sets of files are present at one of the ends of the link beforehand. At first glance this may seem impossible because the calculation of diffs between two files normally requires local access to both files. So how is it possible?

The rsync algorithm takes advantage of the old file. Usually with big files there are only small changes that needs to be updated. Thus the old file will already contain most of the data. Suppose you have two clients that have access to a file A and respectively B, where A and B are "similar". The second client splits the B file into non-overlapping fixed size blocks of data. Than for each block two checksums are calculated and sent to the first client. The first client parses the A file to find the blocks that have the same two checksums. For each founded block, the first client sends back to the second client, a reference to a block in B. The rest of the data, that did not match any of the blocks, is literally sent over the link to the second client. Based on this information, the second client is able to reconstruct the file. A much more detailed description of the rsync algorithm is given in [15]

2.4.3 Crash recovery

Just like AFS and unlike NFS, Samba (CIFS) is a stateful file sharing protocol. Instead of simply processing requests as they come along, CIFS servers keep track of the state of each client.

2.4.4 Security

Samba's built in IP address restrictions may be used with different parameters in order to deny given IP addresses to connect to the server:

- interfaces: using this parameter you can tell Samba to only listen to particular network interfaces

- bind interfaces only: has to be used in conjunction with the *interfaces* parameter. By setting its value to *Yes* access to the interfaces specified via the interfaces parameter is restricted. The nmbd server rejects access attempts based on the client's IP address.
- hosts allow: defines IP addresses of computers that may access the shared data on server
- hosts deny: defines IP address of computer that may NOT access the shared data on server.

The usual way of operation for SMB/CIFS, and thus for Samba, is to pass all the passwords over the network unencrypted. There are solutions available for encrypting Samba traffic[24]. Details about these solutions are beyond the scope of this paper.

The smbd daemon handles "share mode" and "user mode" authentication and authorization. That is, you can protect shared file and print services by requiring passwords. In share mode, the simplest and least recommended scheme, a password can be assigned to a shared directory or printer (simply called a "share"). This single password is then given to everyone who is allowed to use the share. With user mode authentication, each user has their own username and password and the System Administrator can grant or deny access on an individual basis. Linux and Samba passwords are kept in sync via the *passwd* and *smbpasswd* comands. User level offers a convenient way of maintaining and modifying a list of trusted clients without having to reassign a password to a resource each time this list has to change.

2.4.5 System management

Samba includes a management tool called the Samba Web Administration Tool (SWAT). SWAT uses integral samba components to locate parameters supported by the particular version of Samba. It stores the parameter settings, so when SWAT writes the smb.conf file to disk, it will write only those parameters that are different from the default settings. The result is that all comments, as well as parameters that are no longer supported, will be lost from the smb.conf file. Additionally, the parameters will be written back in internal ordering.

The Password Change page is a popular tool that allows the creation, deletion, deactivation, and reactivation of MS Windows networking users on the local machine. Alternately, you can use this tool to change a local password for a user account.

When logged in as a non-root account, the user will have to provide the old password as well as the new password (twice). When logged in as root, only the new password is required.

Chapter 3

Experiment setup

When comparing distributed filesystems, performance is a significant issue. Suppose you want to extend your company and it's time for you to start thinking about a distributed filesystem to use. Or you want to migrate from one filesystem to another. But you would like to have more than just source code and an administrative overview to go on before deciding whether to spend time deploying and learning a filesystem or not; whether spending the time to develop a feel for its performance is worthy or not. This chapter provides empirical data to help decide which filesystem best suits your needs in your computing environment.

3.1 The testing environment

All the servers have the same capabilities:

- CPU type: AMD Athlon(tm) Xp; CPU Speed: 1667 MHz; RAM: 512 MB; OS: SuSE9.1; SWAP space: 1GB; OpenAFS software: openafs-server-1.3.78; Samba software: NFS: NFS version 3.

Linux client:

- CPU type: AMD Sempron(tm) Xp; CPU Speed: 1000 MHz; RAM: 256 MB; OS: SuSE9.1; SWAP space: 1GB; OpenAFS software: openafs-1.3.78; Samba software: NFS: NFS version 3.

client1:

- Machine Type: Acer Ferrari 3000 Series laptop; CPU type: mobile Athlon(tm) Xp; CPU Speed: 2500+; RAM: 512 MB; OS: Windows XP Professional Service Pack 1; OpenAFS software: OpenAFSforWindows-1-3-8400; Samba software: none; NFS software: Microsoft Services for UNIX 3.5 - Client for NFS

Old client:

- Machine type: CINET notebook; CPU type: Pentium 3; CPU Speed: 1000 MHz; RAM: 128 MB; OS: Windows 2000 Professional Service Pack 4; OpenAFS software: OpenAFSforWindows-1-3-8400; Samba software: none; NFS software: Microsoft Services for UNIX 3.5 - Client for NFS

Client2:

- CPU type: AMD Athlon(tm) Xp; CPU Speed: 1000 MHz; RAM: 512 MB; Windows XP professional Service Pack 1; OpenAFS software: OpenAFSforWindows-1-3-8400; Samba software: none; NFS software: Microsoft Services for UNIX 3.5 - Client for NFS

The network diagram in fig3.1 shows how the devices are connected through a Gigabit Ethernet network.

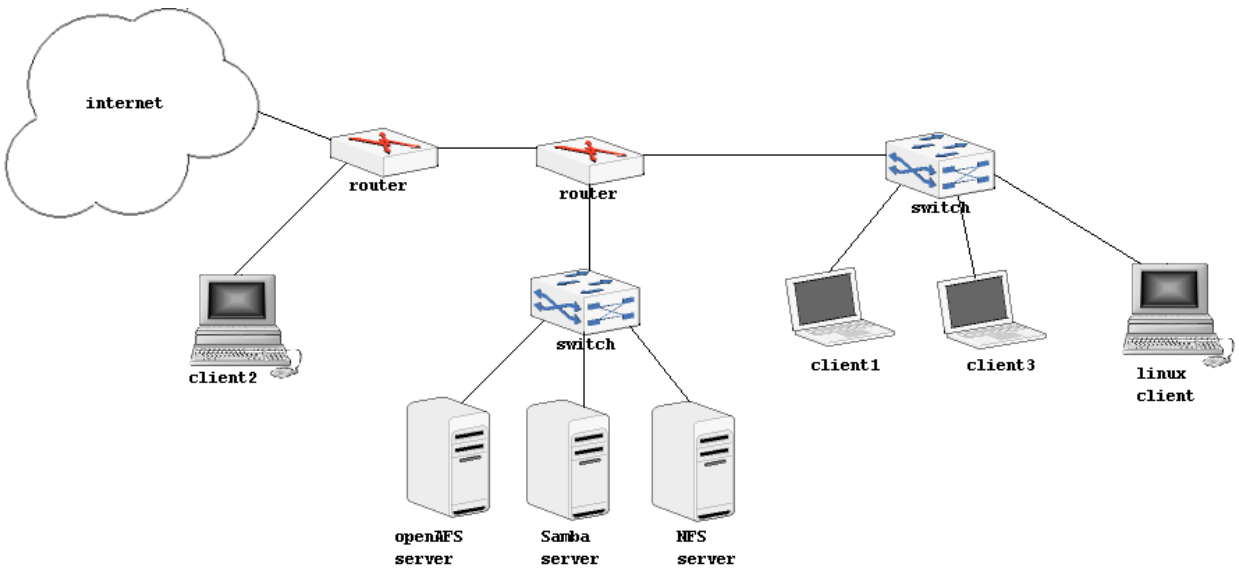


Figure 3.1:Network map

As we've seen the clients, running different OSs, are equipped with different hardware. The servers are equipped with exactly same hardware and are running the same OS. This had to be a requirement in order to be able to compare the filesystems performance behavior.

Note: *different workstations with different hardware and OSs behave different.*

The scope of having clients equipped with different hardware and OSs is to see how these three filesystems performs under different circumstances.

3.2 Data gathering technique

This section discusses some of the available filesystem benchmark tools, which of them was used in collecting the raw performance data and why, and how the experiment was set up.

A benchmark is specially designed program to provide measurements of a particular OS or application.

Available filesystem benchmark tools

Bonnie benchmark measures the performance of hardware and Unix file system trying at the same time to identify performance bottlenecks. It performs sequential output, input and random seeks, on a file of known size.

IOzone benchmark is another tool for measuring a file system's performance by generating several file operations (e.g. read/re-read, write/re-write and so on). The benchmark has been ported to many OS.

The *Postmark benchmark* is an interactive tool designed to measure performance of a distributed filesystem. The benchmark creates a specified number of files of random sizes and afterwards it performs a specified number of transaction on these files. Each transaction consists of sub-transactions that either create, delete, read from or append to files.

With the thought of increasing the accurately of the performance tests values the paper used IOzone benchmark to collect the performance data, even

though the intensive read/write tests of filesystem performance in the Bonnie and IOzone aren't particulary relevant for distributed filesystems.

This was due to the Postmark benchmark's interactive mode making difficult for the analyst to write a script that could perform several consecutive performance tests. Since Bonnie is not ported to other OSs using it could not be an option.

Note: *the chance to get a more accurately true performance value increases with increased number of repeated tests.*

Performance tests set up

IOzone performs many I/O performance tests for several operations, but the paper only takes in consideration the read and write performance. The write test measured the performance of writing a new file in distributed filesystem space. The file sizes tested were: 1 MB, 4 MB, 16 MB, 256 MB, 512 MB and 1 GB. Each file was created using a record size (the amount of data written into a file during a single IO operation) of 4KB. For each file size the standard test was repeated every 5 minutes several times by using a shell script for unix client and a batch script for windows clients (see appendix A). The test's results were directed to a file which later on was parsed by another shell script to write the data to a new files on the form X,Y (see appendix A). Such files are easy to use when creating graphs with xmgrace. The tests were performed on already mounted points (IOzone has the ability to unmount and remount the mount points between tests[30]). The clients are equipped with different hardware and operating system. When these to variables become constant, the benchmark provides a good deal of information about the speed at which different filesystems perform.

Chapter 4

Write performance

The first section of this chapter describes how each filesystem performs on each client, while the second part focuses on the performance comparison of OpenAFS, NFS and Samba on each client.

4.1 Filesystems performance description on each client

This section of the chapter focuses on the write performance describing the rates at which each distributed filesystem performed on each client. It then provides graphs showing the results of running the benchmark on each type of distributed filesystem on the same client. And finally compares and discusses those results.

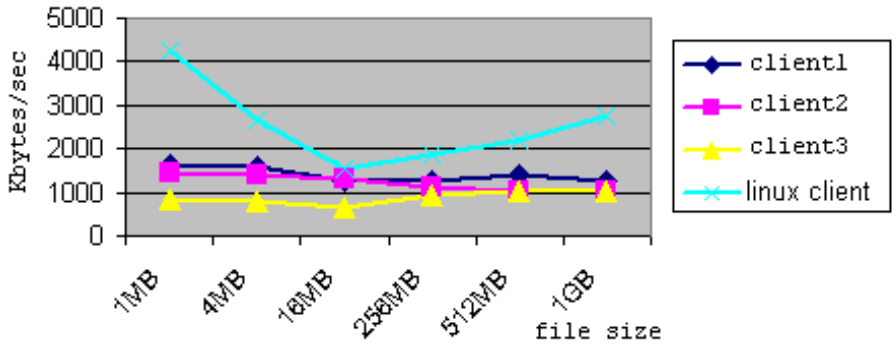


Figure 4.1:OpenAFS average write performance. The graph shows the average write performance for each file size tested. The overlaid lines reflects the clients.

From figure 4.1 we can see that OpenAFS has the poorest performance on old client and the best performance on *linux client*. On each case the performance up to files of 16MB is decreasing with increased file size. As you can see on *linux client* OpenAFS has a considerably decreasing performance, with a performance on 1MB file twice as higher as the performance on 4MB file, and almost three times better than the performance on 16 MB file. For files beyond 16MB, the performance increases with increased file size. On acer and client2 OpenAFS has a quite stable performance with a rate between 1000 Kbytes/sec and 1700 Kbytes/sec, and quite similar to the performance on old client. For file sizes of 16 MB, OpenAFS reaches the "same" performance on all windows clients.

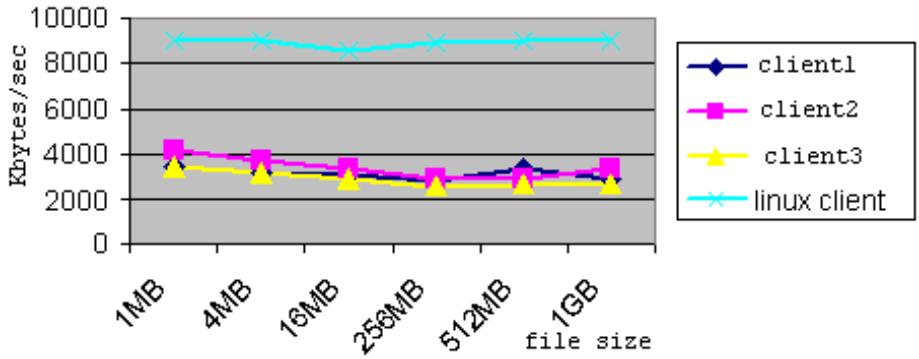


Figure 4.2:NFS average write performance. The graph shows the average write performance for each file size tested. The overlaid lines reflects the clients.

Figure 4.2 shows that NFS achieves best performance on *linux client*; twice as well as the performance on windows clients for all file size. The average performance seems to be quite "constant" an all clients, with not too much differences on the performance rates on windows clients.

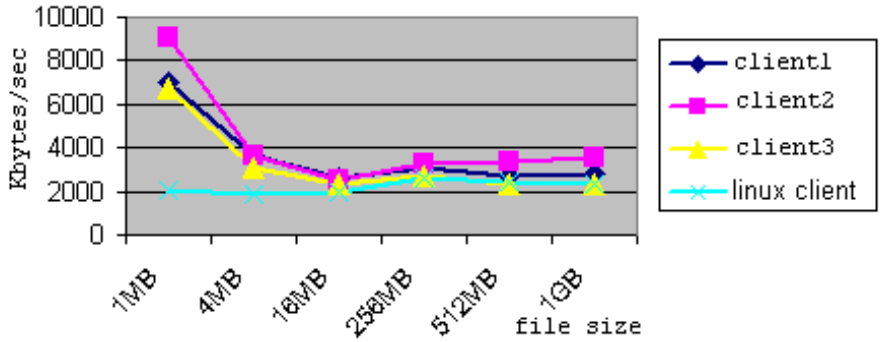


Figure 4.3:Samba average write performance. The graph shows the average write performance for each file size tested. The overlaid lines reflects the clients.

The graph given in figure 4.3 shows that samba has the best average performance on client2, and the poorest on the *linux client*. The average write performance rate for file sizes beyond 16 MB lies between 2000 Kbytes/sec and 4000 Kbyes/sec on all clients. The performance on windows clients is decreasing for file sizes between 1 and 16 MB.

4.2 Performance comparison of OpenAFS, NSF and Samba on each client

After describing how each filesystem behaves on each client, we should turn our focus on how they behave for the same file size an the same client.

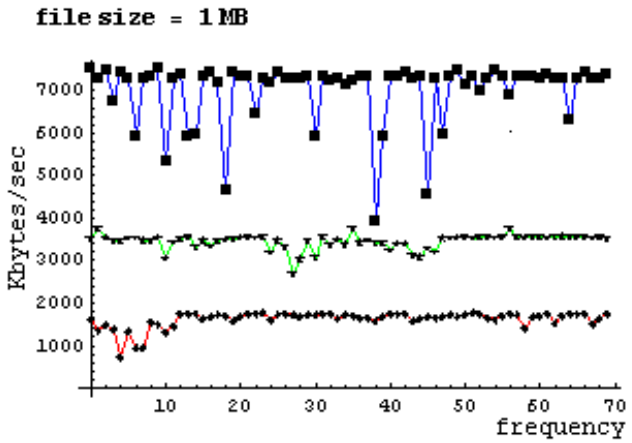


Figure 4.4:Write performance. The graph shows the write performance for file of 1 MB on client1. The overlaid lines represents the filesystems.

It is clear from figure 4.4 that Samba, on client1, has the best performance. Twice as good as NFS and four times better than OpenAFS. But it's performance variations are much higher than both NFS and OpenAFS. OpenAFS and NFS shows a quite stable performance with smaller variations. For each file size (and each client), similar graphs are given in appendix B. Basen on them, we can say that:

- on client1:
  - even though samba's total performance for files of 4 MB is twice lower than its performance for 1 MB file, it still performs better than both NFS and OpenAFS.
  - for files beyond 4 MB, NFS takes the lead performing better than both Samba and OpenAFS.
  - OpenAFS shows the poorest performance for each file size tested.
- on client2:
  - OpenAFS shows the lowest performance for each file size.
  - Samba's shows the same behavior as on client1 for files of 1 MB.
  - With not too much rate differences, NFS shows a lower performance than Samba, except on files of 16 MB, when NFS takes the lead.
- on client3:
  - OpenAFS, once again, performs poorest for each file size tested.
  - On files of 4 MB, for both NFS and Samba the benchmark tool gives quite similar rate values. But Samba still has higher variations in its performance.
  - On files beyond 16 MB NFS takes the lead for both performance and variations in performance.
- on *linux client*:
  - NFS performs better than both Samba and OpenAFS. Its performance variations are not too high for neither different file sizes nor on performance on each file. Hence, it keeps a stable performance.

– Samba shows the poorest performance for files bellow 16 MB. Beyond this edge it performs better than OpenAFS.

4.3 Summery

Data shown in graphs given in section and appendix B reveals several important differences between these three filesystems. As we've seen, both NFS and Samba performed twice to three times as well as AFS during the writing test, on windows clients. Their performance rates with file sizes greater than 1MB are almost on the same scale. Best performance for Samba was for the file sizes of 1 MB. For the file sizes beyond 1 MB, Samba performance drops considerably. Even though NFS and OpenAFS performance varies, we could say that both of them do maintain a "constant" performance for all file sizes. On the *linux client*, NFS is the winner. It performs twice as well as AFS and four times as well as Samba for the file size of 1MB. For file sizes over 1MB, NFS performs three to four times as well as both Samba and OpenAFS. OpenAFS has the lowest performance on all clients.

Chapter 5

Read performance

The first section of this chapter describes how each filesystem performs on each client, while the second part focuses on the performance comparison of OpenAFS, NFS and Samba on each client.

5.1 Filesystems performance description on each client

This section of the chapter focuses on the read performance describing the rates at which each distributed filesystem performed on each client. It then provides graphs showing the results of running the benchmark on each type of distributed filesystem on the same client. And finally compares and discusses those results.

Note: *all the average values used to plot the graphs in both this and next chapter were calculating using Microsoft Excel. They are to be found in appendix E.*

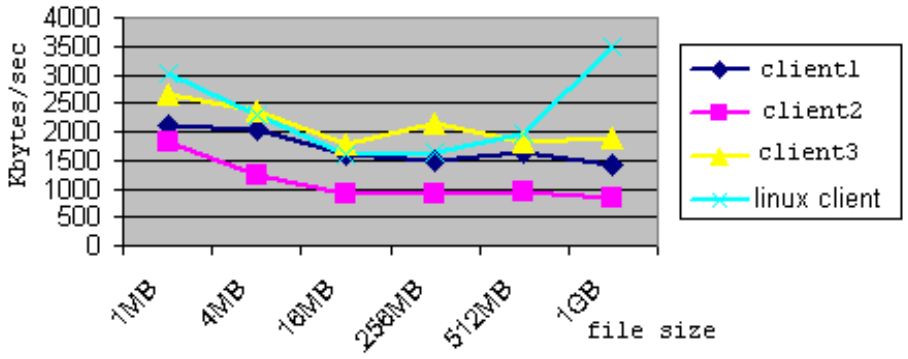


Figure 5.1:OpenAFS average read performance. The graph shows the average read performance for each file size tested. The overlaid lines reflects the clients.

From figure 5.1 we can see that OpenAFS achieve the poorest performance on client2 and the best performance on *linux client*. On each case the performance up to files of 16MB is decreasing with increased file size. As you can see on *linux client* OpenAFS has its best average performance rate for files of 1 GB. Its average performance for file sizes between 16 and 512 MB is almost the same for old, acer and *linux client*. The client3 makes the exception by showing a increased average performance rate on files of 256 MB.

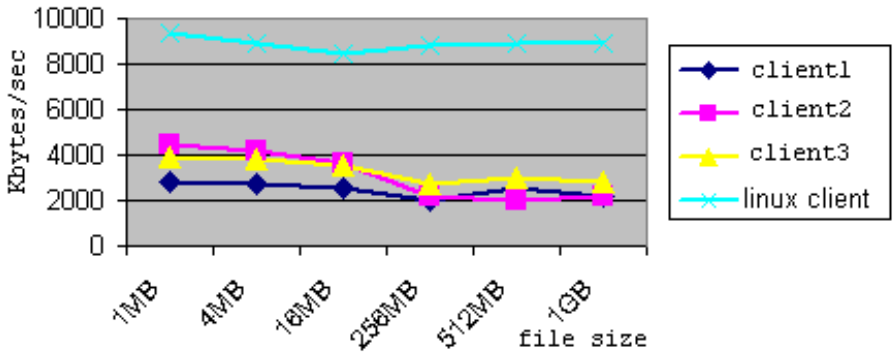


Figure 5.2:NFS average read performance. The graph shows the average read performance for each file size tested. The overlaid lines reflects the clients.

From figure 5.2 we see that NFS has twice as better performance on *linux client* than on windows clients. Its performance behavior on windows clients is quite similar: decreasing rates with increased file sized up to 256 MB. Beyond this edge the performance remains quite stable, except on client1 that shows a little bit better average performance on file of 512 MB.

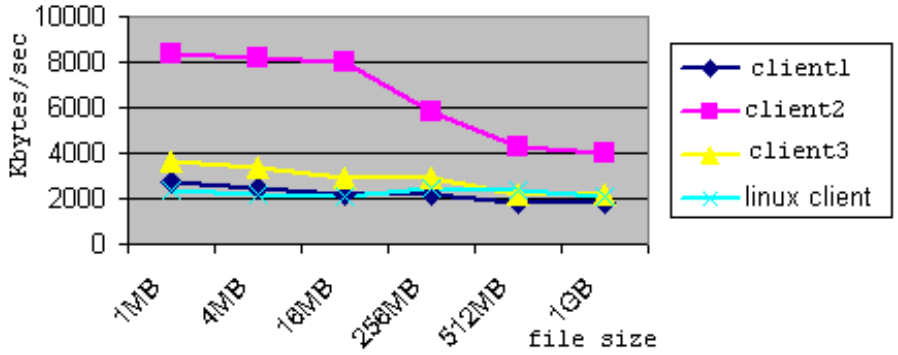


Figure 5.3:Samba average read performance. The graph shows the average read performance for each file size tested. The overlaid lines reflects the clients.

Figure 5.3 shows an interesting behavior for Samba performance. It performs similar on *linux client*, client3 and client2, with rates between 2000 Kbytes/sec and 4000 Kbytes/sec. The interesting thing is that it performs much better on client1, with a rate twice as high for almost all file sizes tested. Unfortunately due to lack of time this phenomena is not analyzed in this paper. One explanation could maybe be the differences in clients architectures. Client1 has a memory size of 512 MB, client3 128 MB and *linux client* 256 MB. Their CPU speeds differ as well. Even though they all run different OS and are equipped with different hardware, they do perform almost equally.

Do the other clients run too many background processes requiring a lot of CPU time leading to CPU performance bottleneck? Client3 is a three - four years old laptop that hasn't been used for at least two years. Not before starting on this performance comparison experiment, when I did a clean install on it and other than that just the required software for the experiment. So, too many background process could'n be running on it. Client1 is a two years old laptop that hasn't been maintained at all. During these two years I have installed a lot of software, much of them are even forgotten that they are there. So the hypothesis could maybe be true. *Linux client* is a brand new machine running SuSE9.1. No other software then the required ones, for the experiment, are installed on it. Thus, the hypothesis has a really low probability to be true for this client as well. What about paging activity? Or really idle CPU time waiting for I/O request to finish? Finding answers is not the goal of this paper, but definitely a recommendation for further work.

5.2 Performance comparison of OpenAFS, NSF and Samba on each client

Now that we've seen how each filesystem behaves on each client, we are ready to analyze and compare the performance for each file size on each client.

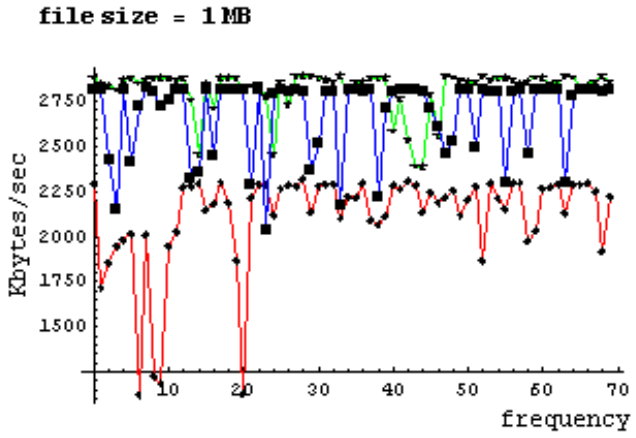


Figure 5.4:Read performance. The graph shows the read performance for file of 1 MB on client1. The overlaid lines represents the filesystems.

As shown in figure 5.4 NFS and Samba have the same performance rate and a little bit higher than OpenAFS. While OpenAFS improves its read performance compared to its write performance, NFS and Samba have a lower read performance. For each file size tested, on each client, similar graphs are given in appendix B. Based on them, we can say that:

- on client1:
  - OpenAFS has the lowest total performance for each file size tested. Unlike its write performance, there are high variations in its read performance for each file size tested. Hence, the performance is not stable.
  - for file sizes beyond 4 MB NFS has the best performance.
  - none of the filesystem performs stable for any of the file sizes tested.
- on client2:

  - Samba has the best total performance for each file size, its performing rate for files up to 16 MB being on the same scala, while for files beyond that its performance variations increases with increased file size.
  - OpenAFS shows a stable performance for all file sizes.
  - for each file size NFS performs twice as bad as Samba and almost twice as good as OpenAFS.

- on client3:
  - NFS has the best total performance for each file size.
  - OpenAFS has the poorest total performance for each file size.

- on *linux client*:
  - NFS performs at least twice as good as both OpenAFS and Samba for each file size tested.
  - Performance rate for both OpenAFS and Samba are quite similar, except for 1MB file where Samba shows a little bit higher rate than OpenAFS, and for file of 1 GB where OpenAFS has a better performance than Samba.

5.3 Summary

As we have seen, both NFS and Samba performed much better than AFS during the reading test, on windows clients. On client2 Samba's read performance is twice as good as its write performance, and much higher than both NFS and OpenAFS. NFS performs best on *linux client* and as well as Samba on client2.

Chapter 6

Statitstics

In previous sections we discussed how well OpenAFS, NFS and Samba performed. Our descriptions were based on observational estimations. In this chapter we describe how we can make use of statistics to find out the true measurement value. Due to too much available data, we will be using only one of our data sets: measurements of openAFS's write performance on *old client* for a file of 4 MB (randomly chosen data set).

6.1 Frequency distribution

The repeated measurements gives us the data set given bellow:

642, 530, 654, 599, 680, 644, 544, 611, 617, 621, 496, 411, 463, 431, 570, 900, 678, 396, 468, 441, 443, 468, 416, 388, 458, 369, 429, 725, 1253, 1326, 848, 481, 586, 756, 1363, 1370, 1193, 826, 521, 469, 727, 980, 1140, 1266, 586, 538, 596, 1290, 1379, 1376, 1384, 1355, 1116, 424, 484, 1193, 1375, 1314, 822, 596, 538, 600, 1216, 1389, 1374, 1364, 946, 524, 492, 513

So far this sample does not tell us anything. What we are interested in, is the shape of this distribution. Why? If the distribution is symmetrical about its mean value we can probably state that the true value of the measurements is actually the mean value. If the distribution is asymmetrical, we need deeper investigation to find some explanation.

.

Note: *The histogram in fig6.1 was created with Microsoft Excel.*

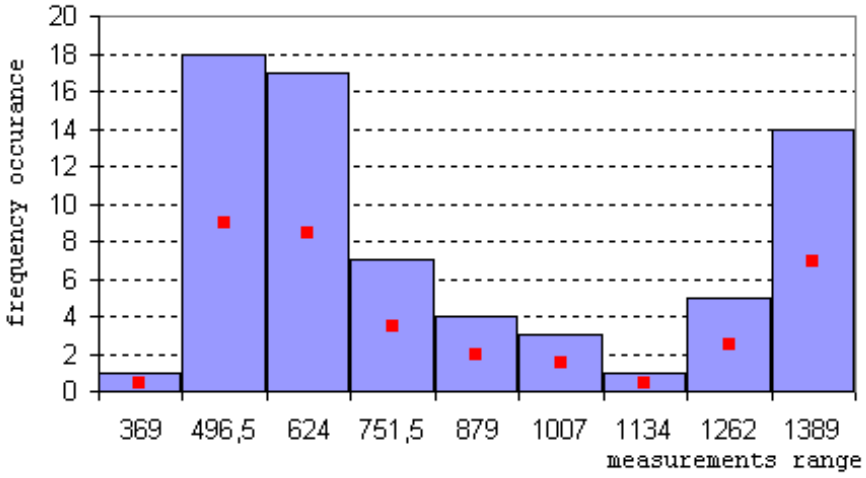


Figure6.1 Frequency distribution

As the shape of the distribution shows we deal with a asymmetrical distribution. Thus, we can say that OpenAFS's performance for files of 16 MB on *old client* changes its performance with a constant probability of unit time.

Note: *This paper is not intended for any deeper investigation. It could be interesting finding out an explanation to this behavior and therefor recommended for further work.*

6.2 Uncertainty

When, reporting test results you should also report their measurement uncertainty. *'The measurement uncertainty of a test is as important as the test result itself'* [10].

The fault tree diagram given in figure 6.1 shows different sources of uncertainty.

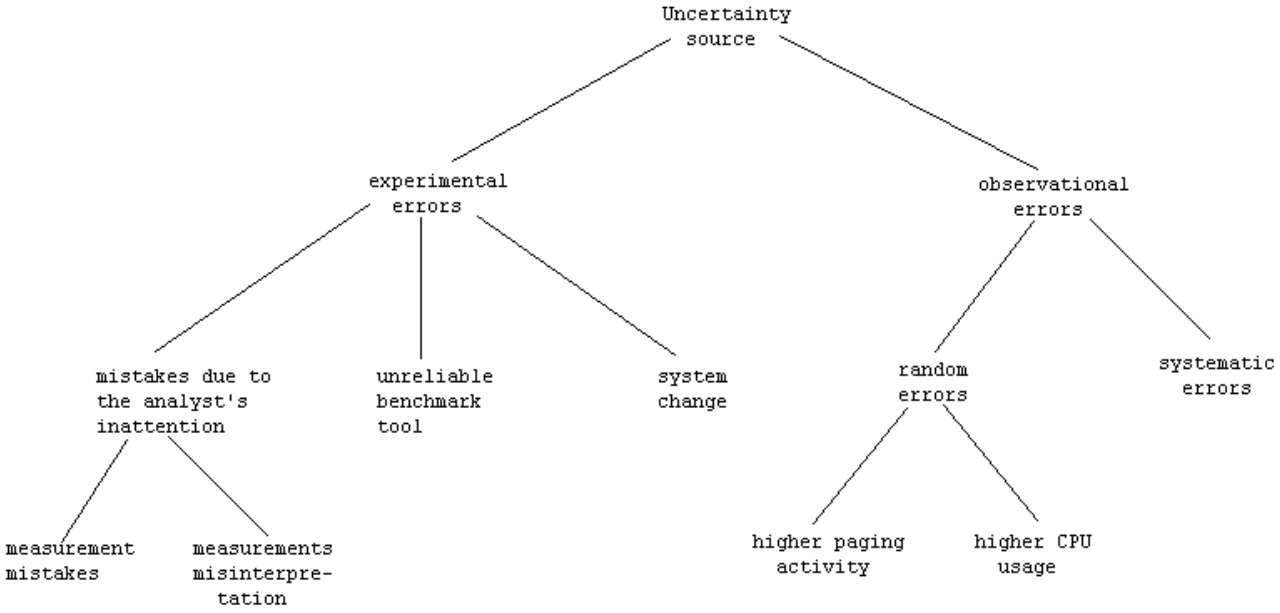


Figure 6.1: Uncertainty fault tree

Thus the question we should ask here is how accurate these measurements are. As we've seen our distribution is not a normal distribution. The frequency distribution shape is bimodal (higher peak at the left and a long tail with a new high peak on the right). However, '*decades of artificial courses on statistics have convinced many scientists that the distribution of points about the mean must follow a Gaussian normal distribution in the limit of large numbers of measurements*' [3].

Based on the idealized limit of an infinite number of points, *Uncertainty*, is expressed as:

$$P(\chi_i) = \frac{1}{(2\pi\sigma^2)^{\frac{1}{2}}} \exp\,(-\frac{(\chi_i - \bar{\chi})^2}{2\sigma^2}) \tag{6.2.1}$$

$Where,$

$P(\chi_i)$= the probability of our expectation for the measurement value to be distributed about the mean value
σ= the standard deviation of the data set
χ_i= the value of the ith X in the sample
$\bar{\chi}$= the mean value

Standard deviation is expressed as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N} \Delta g_i^2} \tag{6.2.2}$$

with N and Δg_i defined as: N = number of repeated measurements

and

Δg_i= error in the measurement, defined by:

$$\Delta g_i = \langle \text{arithmetic mean of the data} \rangle - \chi_i \tag{6.2.3}$$

Note : formulaes used in this section are defined in [3] chapter3.

Now that we have defined all the values, we shall use a randomly chosen value from our data set to calculate its probability distribution about the mean.

```
In[1]:= <<Statistics`DescriptiveStatistics`

In[2]:= measurement values list

In[3]:= data1 = {642, 530, 654, 599, 680, 644, 544, 611, 617, 621, 496, 411,
        463, 431, 570, 900, 678, 496, 468, 441, 443, 468, 416, 388, 458,
```

```
    369, 429, 725, 1253, 1326, 848, 481, 586, 756, 1363, 1370, 1193,
    826, 521, 469, 727, 980, 1140, 1266, 586, 538, 596, 1290, 1379,
    1376, 1384, 1355, 1116, 424, 484, 1193, 1375, 1314, 822, 596, 538,
    600, 1216, 1389, 1374, 1364, 946, 524, 492, 513}
```

Out[3]= {642, 530, 654, 599, 680, 644, 544, 611, 617, 621, 496, 411, 463, 431, 570, 900, 678, 496, 468, 441, 443, 468, 416, 388, 458, 369, 429, 725, 1253, 1326, 848, 481, 586, 756, 1363, 1370, 1193, 826, 521, 469, 727, 980, 1140, 1266, 586, 538, 596, 1290, 1379, 1376, 1384, 1355, 1116, 424, 484, 1193, 1375, 1314, 822, 596, 538, 600, 1216, 1389, 1374, 1364, 946, 524, 492, 513}

mean (arithmetic mean)

In[4]:=
```
Mean[data1]
```

Out[4]= $\frac{55081}{70}$

In[5]:=
```
data2 = (Mean[data1] - 642)^2 + (Mean[data1] - 530)^2 + (Mean[data1] - 654)^2 +
  (Mean[data1] - 599)^2 + (Mean[data1] - 680)^2 + (Mean[data1] - 644)^2 +
  (Mean[data1] - 544)^2 + (Mean[data1] - 611)^2 + (Mean[data1] - 617)^2 +
  (Mean[data1] - 621)^2 + (Mean[data1] - 496)^2 + (Mean[data1] - 411)^2 +
  (Mean[data1] - 463)^2 + (Mean[data1] - 431)^2 + (Mean[data1] - 570)^2 +
  (Mean[data1] - 900)^2 + (Mean[data1] - 678)^2 + (Mean[data1] - 396)^2 +
  (Mean[data1] - 468)^2 + (Mean[data1] - 441)^2 + (Mean[data1] - 443)^2 +
  (Mean[data1] - 468)^2 + (Mean[data1] - 416)^2 + (Mean[data1] - 388)^2 +
  (Mean[data1] - 458)^2 + (Mean[data1] - 369)^2 + (Mean[data1] - 429)^2 +
  (Mean[data1] - 725)^2 + (Mean[data1] - 1253)^2 + (Mean[data1] - 1326)^2 +
  (Mean[data1] - 848)^2 + (Mean[data1] - 481)^2 + (Mean[data1] - 586)^2 +
  (Mean[data1] - 756)^2 + (Mean[data1] - 1363)^2 + (Mean[data1] - 1370)^2 +
  (Mean[data1] - 1193)^2 + (Mean[data1] - 826)^2 + (Mean[data1] - 521)^2 +
  (Mean[data1] - 469)^2 + (Mean[data1] - 727)^2 + (Mean[data1] - 980)^2 +
  (Mean[data1] - 1140)^2 + (Mean[data1] - 1266)^2 + (Mean[data1] - 586)^2 +
  (Mean[data1] - 538)^2 + (Mean[data1] - 596)^2 + (Mean[data1] - 1290)^2 +
  (Mean[data1] - 1379)^2 + (Mean[data1] - 1376)^2 + (Mean[data1] - 1384)^2 +
  (Mean[data1] - 1355)^2 + (Mean[data1] - 1116)^2 + (Mean[data1] - 424)^2 +
```

```
    369, 429, 725, 1253, 1326, 848, 481, 586, 756, 1363, 1370, 1193,
    826, 521, 469, 727, 980, 1140, 1266, 586, 538, 596, 1290, 1379,
    1376, 1384, 1355, 1116, 424, 484, 1193, 1375, 1314, 822, 596, 538,
    600, 1216, 1389, 1374, 1364, 946, 524, 492, 513}
```

Out[3]= {642, 530, 654, 599, 680, 644, 544, 611, 617, 621, 496, 411, 463, 431, 570, 900, 678, 496, 468, 441, 443, 468, 416, 388, 458, 369, 429, 725, 1253, 1326, 848, 481, 586, 756, 1363, 1370, 1193, 826, 521, 469, 727, 980, 1140, 1266, 586, 538, 596, 1290, 1379, 1376, 1384, 1355, 1116, 424, 484, 1193, 1375, 1314, 822, 596, 538, 600, 1216, 1389, 1374, 1364, 946, 524, 492, 513}

mean (arithmetic mean)

In[4]:= **Mean[data1]**

Out[4]= $\frac{55081}{70}$

```
In[5]:= data2 = (Mean[data1] - 642)^2 + (Mean[data1] - 530)^2 + (Mean[data1] - 654)^2 +
          (Mean[data1] - 599)^2 + (Mean[data1] - 680)^2 + (Mean[data1] - 644)^2 +
          (Mean[data1] - 544)^2 + (Mean[data1] - 611)^2 + (Mean[data1] - 617)^2 +
          (Mean[data1] - 621)^2 + (Mean[data1] - 496)^2 + (Mean[data1] - 411)^2 +
          (Mean[data1] - 463)^2 + (Mean[data1] - 431)^2 + (Mean[data1] - 570)^2 +
          (Mean[data1] - 900)^2 + (Mean[data1] - 678)^2 + (Mean[data1] - 396)^2 +
          (Mean[data1] - 468)^2 + (Mean[data1] - 441)^2 + (Mean[data1] - 443)^2 +
          (Mean[data1] - 468)^2 + (Mean[data1] - 416)^2 + (Mean[data1] - 388)^2 +
          (Mean[data1] - 458)^2 + (Mean[data1] - 369)^2 + (Mean[data1] - 429)^2 +
          (Mean[data1] - 725)^2 + (Mean[data1] - 1253)^2 + (Mean[data1] - 1326)^2 +
          (Mean[data1] - 848)^2 + (Mean[data1] - 481)^2 + (Mean[data1] - 586)^2 +
          (Mean[data1] - 756)^2 + (Mean[data1] - 1363)^2 + (Mean[data1] - 1370)^2 +
          (Mean[data1] - 1193)^2 + (Mean[data1] - 826)^2 + (Mean[data1] - 521)^2 +
          (Mean[data1] - 469)^2 + (Mean[data1] - 727)^2 + (Mean[data1] - 980)^2 +
          (Mean[data1] - 1140)^2 + (Mean[data1] - 1266)^2 + (Mean[data1] - 586)^2 +
          (Mean[data1] - 538)^2 + (Mean[data1] - 596)^2 + (Mean[data1] - 1290)^2 +
          (Mean[data1] - 1379)^2 + (Mean[data1] - 1376)^2 + (Mean[data1] - 1384)^2 +
          (Mean[data1] - 1355)^2 + (Mean[data1] - 1116)^2 + (Mean[data1] - 424)^2 +
```

```
(Mean[data1] - 484)^2 + (Mean[data1] - 1193)^2 + (Mean[data1] - 1375)^2 +
(Mean[data1] - 1314)^2 + (Mean[data1] - 822)^2 + (Mean[data1] - 596)^2 +
(Mean[data1] - 538)^2 + (Mean[data1] - 600)^2 + (Mean[data1] - 1216)^2 +
(Mean[data1] - 1389)^2 + (Mean[data1] - 1374)^2 + (Mean[data1] - 1364)^2 +
(Mean[data1] - 946)^2 + (Mean[data1] - 524)^2 + (Mean[data1] - 492)^2 +
(Mean[data1] - 513)^2
```

Out[5]= $\frac{609632949}{70}$

standard deviation

In[6]:= $\text{stdDev} = \sqrt{\frac{1}{70} \times \text{data2}}$

Out[6]= $\frac{\sqrt{609632949}}{70}$

the probability of randomely chosen values from our data set

$$P = \frac{1}{(2 \times 3.14 \times \text{stdDev}^2)^{\frac{1}{2}}} \times \text{Exp}\left[-\frac{(\text{data3} - \text{Mean[data1]})^2}{2 \times \text{stdDev}^2}\right]$$

In[8]:= `data3 = 725`

Out[8]= 725

In[9]:= `P`

Out[9]= 0.00111404

In[10]:= `data3 = 599`

Out[10]= 599

In[11]:= `P`

Out[11]= 0.000981704

From the results that mathematica gives us, we can say that the probability is very low for each of the measurement value. In other words the distribution of the values measured, in our case, is not due to random error. Finding the cause behind them is recommended for further work.

Chapter 7

Conclusion

There are many distributed filesystems, each of which is appropriate for a particular environment. If you just want to begin sharing volumes over a network, NFS is obviously the best choice. On machines running linux, it performs at a much higher rate than both Samba and OpenAFS. On machines running windows it performs at an equal rate with Samba's. Their performance being higher than OpenAFS's performance. If you just want your windows users to be able to access linux shares, than Samba is better than NFS. It performs as well as NFS on windows, worse on linux machines, but does offer a better solution to security. If you want to guarantee your users higher availability, want greater protection from data loss during a crash, and want a transparent, unified namespace with minimal configuration on a client system, OpenAFS is the best choice.

OpenAFS was the most difficult and time consuming distributed filesystem to install; the whole setup process took approximately three weeks, while installing and setting up NFS and samba required just few of hours.

Two main properties of AFS are security and scalability[17], both of which complicate system administration even if all future are not fully utilized. Since OpenAFS uses different semantics than Unix [2], understanding and learning to maintain OpenAFS requires a system administrator much time. Both NFS and Samba was much easier to install and the set up processes took only 30 minutes.

Security mechanism in AFS is more significant than security mechanism in

both NFS and Samba. The authentication mechanism used in AFS, such as the Kerberos key distribution protocol, introduces a high level of protection to the system. Also, the ability to encrypt valuable filesystem information during its transfer over the network makes it a hard target for penetrators[17].

Samba supports stateful servers in a way similar to OpenAFS, while NFS is a stateless server. Both Samba and OpenAFS keep track of all the client's cached and replicated data and maintain consistency among parallel copies. Local server crashes halt both NFS and Samba. When an OpenAFS file server crashes, clients can be switched transparently to a different file server under the same cell, due to its location transparency implementation.

In terms of read performance based on IOzone benchmark, both NFS and Samba, showed a approximately 25% better read performance than Samba on both client1 and client3). On client3 Samba showed a read performance twice as high as NFS for file sizes bellow 256 MB. Above this edge Samba showed a more then 50% better performance than NFS. OpenAFS had a four times lower read performance then Samba for each file size tested. On *linux client* (SuSE9.1), NFS performed twice as well as both OpenAFS and Samba for all file sizes. Samba and OpenAFS performed as well as each other, exception for file of 1 GB where OpenAFS increased its performance.

In terms of write performance based on IOzone benchmark, for *linux client* the filesystems performance results are the same as with read performance. OpenAFS has the lowest write performance on all windows clients. Even though Samba shows a drastically reduced performance from file sizes of 1MB to 4MB, NFS and Samba performed, for file sizes of 4 MB up to 1 GB, comparably on all windows clients.

Thus, on windows clients, OpenAFS had the poorest read/write performance; NFS had a read/write performance equal to that of Samba. On *linux client*, NFS is clearly the winner of both read and write performance, while OpenAFS and Samba performed as well as each other.

Bibliography

[1] AFSLore. http://grand.central.org/twiki/bin/view/afslore.

[2] Eliezer Levy & Silberschatz A. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 1990.

[3] Mark Burgess. *Analytical Network And System Administration*. WILEY, 2004.

[4] Bondarenko B. Comparison and evaluation of nfsv3, nfsv4, and afs distributed filesystems. *Technical Report CMU-CS-93-156*, 1993.

[5] Rosen M.B. Wilde M.J. & Fraser-Campbell B. Nfs portability. *Summer USENIX Conference Proceedings*, 1986.

[6] Sandberg R. Goldberg D. Kleiman S. Walsh D. & Lyon B. Design and implementation of the sun network filesystem. *Summer USENIX Conference Proceedings*, 1985.

[7] Taylor B. Secure networking in the sun environment. *USENIX Association Conference Proceedings*, 1986.

[8] Taylor B. A framework for network security. *SunTechnology 1(2)*, 1988.

[9] Zayas E.R. Everhart C.F. Design and specification of the cellular andrew environment. *Technical Report CMU-ITC-070*, 1988.

[10] Ronald H. Dieck. *Measurement Uncertainty Methods and Applications, third edition*. ISA, 2002.

[11] Steve French. Samba, cifs and linux network filesystems ibm corp.

[12] Howard J.H. An overview of the andrew file system. *Proceedings of the USENIX Winter Technical Conference*, 1988.

[13] Kistler J. J. Disconnected operation in a distributed file system. *Technical Report CMU-CS-93-156*, 1993.

[14] Rosenblum M. & Ousterhout J. The design and implementation of a log-structured file system for unix. *ACM Transactions on Computer Systems*, 1992.

[15] Andrew Tridgell & Paul Mackerras. The rsync algorithm. Technical report, Australian National University, June 1996.

[16] Michael Maclsaac. Samba project documentation ibm corp 2003.

[17] Howard J.H. Kazar M.L. Menees S.G. Nichols D.A. Satyanarayanan M. Sidebotham R.N. & West M.J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[18] Satyanarayanan M. Howard J.H. Nichols D.A. Spector A.Z. & West M.J. The itc distributed file system: Principles and design. *Proceedings of the Tenth ACM Symposium on Operating System Principles*, 1988.

[19] Kazar M.L. Synchronization and caching issues in the andrew file system. *Proceedings of the USENIX Winter Technical Conference*, 1988.

[20] Satyanarayanan M. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7:247–280, 1989.

[21] Sidebotham R.N. Volumes: The andrew file system data structuring primitive. *Mass Storage Systems, Towards Distributed Storage and Data Management Systems*, First International Symposium Proceedings, Thirteen IEEE Symposium, 1986.

[22] Andrew Tridgell & Paul Mackerras The rsync algorithm. http://samba.anu.edu.au/rsync/tech_report/tech_report.html.

[23] Stern H. Eisler M. & Labiaga R. *Managing NFS and NIS, 2nd Edition*. O'Reilly, 2001.

[24] Roderick W. Smith. *The Definitive Guide to Samba 3*. Apress, 2004.

[25] Inc. 1998 Sun Microsystems. Network programming. *Ascend McNair Conference at the Graduate Center of the City University of New York*, 2001.

[26] SAMBA Team. Migrating windows servers to samba.

[27] John H. Terpstra. *Samba-3 by Example: Practical Exercises to Successful Deployment*. Prentice Hall PTR, 2004.

[28] Rainer Többicke. Distributed file systems: Focus on the andrew file system/distributed file system(afs/dfs). *Technical Report CMU-CS-93-156*, 1993.

[29] D. Walsh B. Lyon J. Chang D. Goldberg S. Kleiman T. Lyon R. Sandberg & P. Weiss. Overview of the sun network file system. *USENIX Association Conference Proceedings*, pages 177–124, 1985.

[30] Norcott W. Iozone benchmark. http://www.iozone.org/.

[31] Erez Zadok. *Linux NFS and Automounter Administration*. SYBEX, 2001.

Appendix A

Installing the software

Unlike both NFS and Samba, installing OpenAFS on SuSE was a complex job. Trying to compile the source code was a waist of time. During the this operation with each error message that was fixed, another error message was issued. But the whole process was so interesting for the author making her keep trying for several weeks. It's not a really smart behavior when you have a time limit to complete the experiment. Even though the task is supposed to be possible, the author had to give up and rather install the already compiled rpm packages. Additionally rpm packages had to be installed before everything going smoothly. Prerequisites libraries are: kernel non-gpl (needed by OpenAFS client), kerberos5, afs kerberos, pam kerberos5. The server set up process was too a complicated task to complete. Some guidelines are given bellow:

- make sure the partition you want to export is either ext2 or ext3. OpenAFS on linux does not work with other file systems.
- your /etc/hosts file must contain the right ip address of your server
- after defining your afs cell, make sure your CellServDB file contains the server's ip address and not the loopback address. The author made the silly mistake not doing so. The consequence was an error message that wasn't too easy to understand. Hence, you will be avoiding spending time on finding out what the error message tries to tell you.
- do NOT forget to set DYNROOT="yes" in your /etc/sysconfig/afs-client before installing the client part on the server. This tells OpenAFS that you install your first afs server. Otherwise AFS will not grant you any access to the volume, even though you have the right permissions. Not checking the afs-client file was another mistake the author did. The consequence: waist of valuable time on trying to understand why I didn't have administrator rights on the volume. Foolish mistake when the official documentation notifies about this.
- a final error that the author had to deal with, on *linux client* side, was:

  linuxClient:/home/claudia2 # ls -l /afs
  fs:'/afs': Connection timed out

A quick search on google lead to finding out how to go any further: wait for a few hours and see what happens. I've waited alright! But nothing change. The problem was that the /afs root did not have the right permissions bits. These should be as follows:
afsServer:/home/claudia2 # ls -ld /afs
drwxrwxrwx 2 root root 2048 2005-04-27 00:48 /afs

Lesson learned by the author: read the documentation carefully.

Installing and configuring NFS and Samba was an easy task without any complications.

As mentioned before Microsoft Windows Services for UNIX (SFU) 3.5 is the NFS client software on windows machines. It allows data sharing in a heterogenous network. Its *Client for NFS* is easy to install end configure and enables Windows computers to access resources exported from NFS file servers. It supports both NFS version 2 and 3. Windows users can access NFS exports as they would access a Windows share. Using Windows Explorer and Navigating to "My network places-NFS network-NFS server-share" (fig2.2), NFS exports can be mapped to drive letters.

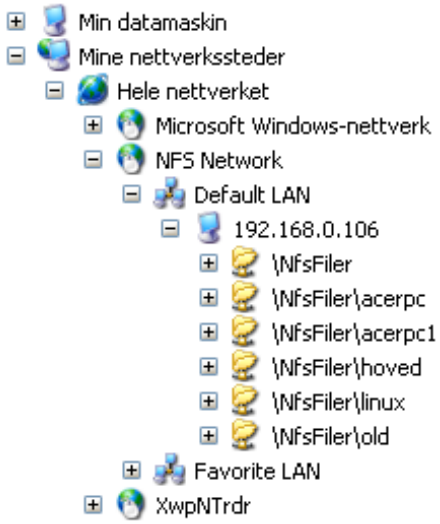


Figure 2.2:SFU view from windows explorer.

Appendix B

Performance measurement scripts

The scripts repeats an IOzone standard measurement several times, with 5 minutes interval. The measurement results are directed to a file. The options to the measurement are:

-s option is the size of the file to be measure

-r is the record size, in Kbytes

-i 0 tells iozone to measure the write performance

-i 1 tells iozone to measure the read performance

-R option generates an Excel report

-c includes *close()* in time calculation

Linux

runTest.sh

```
#! /bin/bash

COUNTER=0
while [ $COUNTER -lt 70 ]; do
       ./iozone -s 1M -r 4k -i 0 -i 1 -Rac >> acerMeasureAFS.txt
       let COUNTER=COUNTER+1
       sleep 5
       echo $COUNTER
done
```

Windows

runTest.bat

```
set var1=0
set var2=0
:while
if %Var1%==7 goto END
call add.bat
# G is the mounted drive letter
G:\acer\iozone -s 1M -r 4k -i 0 -i 1 -Rc >> acerMeasureAFS.wks
sleep 300
goto while
:END
```

add.bat

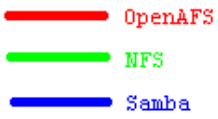
```
:: Increments a two digit number
:: Works by comparing each digit
:: var1=tens, var2=ones
if [%Var1%]==[] set Var1=0
if [%Var2%]==[] set Var2=0
:var2
if %var2%==9 goto var1
if %var2%==8 set var2=9
if %var2%==7 set var2=8
if %var2%==6 set var2=7
if %var2%==5 set var2=6
if %var2%==4 set var2=5
if %var2%==3 set var2=4
if %var2%==2 set var2=3
if %var2%==1 set var2=2
if %var2%==0 set var2=1
goto DONE
:var1
set var2=0
if %var1%==9 goto DONE
if %var1%==8 set var1=9
if %var1%==7 set var1=8
if %var1%==6 set var1=7
if %var1%==5 set var1=6
if %var1%==4 set var1=5
if %var1%==3 set var1=4
if %var1%==2 set var1=3
if %var1%==1 set var1=2
if %var1%==0 set var1=1
goto DONE
:DONE
```

```
#! /usr/bin/perl

#the script parses the output file from IOzone
#and writes the read/write measurement rate to
#two different files. One that can be used with
#mathematica and another one for use with Microsof
#Excel. The graphs shwon in appendix B and C are
#generated with Mathematica. The average and standard
#deviation tables shwon in appendix E are generated
#with Microsoft Excel.

$teller = 0;

open (FIL, "./AFS1GLinux.txt") or
die "cant open file /AFS1GLinux.txt\n";

open (WRITER, ">>afsWriterLinux.txt") or die \\
"cant open file afsWriterLinux.txt\n";

open (WRITER1, ">>afsWriterExcelLinux.txt") or die \\
"cant open file afsWriterExcelLinux\n";

open (READER, ">>afsReaderLinux.txt") or die \\
"cant open file afsReaderLinux.txt\n";

open (READER1, ">>afsReaderExcelLinux.txt") or die \\
"cant open file afsReaderExcelLinux\n";

while($line = <FIL>)
  {
    if($line =~ /^"Writer report"/)
      {
        $line1 = <FIL>;  #reads a line of no interrest
  #line containing the files size and Mbps value
        $line1 = <FIL>;
        @array = split(" ",$line1); #splits the line
  #writes the Mbps value to the file
        print WRITER1 "$array[1]\n";
```

```
        print WRITER "{$teller,";
        print WRITER "$array[1]},";
        $teller += 1;
      }
    if($line =~ /^"Reader report"/)
      {
        $line1 = <FIL>;
        $line1 = <FIL>;
        @array = split(" ",$line1);
        print READER1 $array[1];
        print READER1 "\n";
        print READER "{$teller,";
        print READER "$array[1]},";

      }

  }

close FIL;
close WRITER;
close WRITER1;
close READER;
close READER;
```

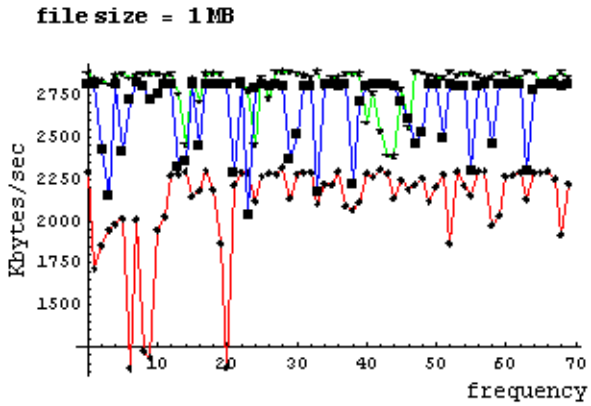
Appendix C

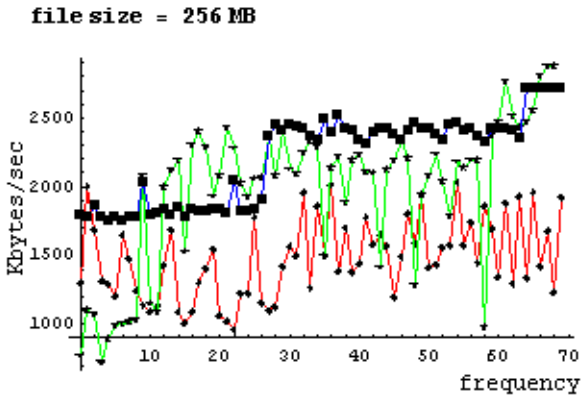
Overlaid lines read performance graphs

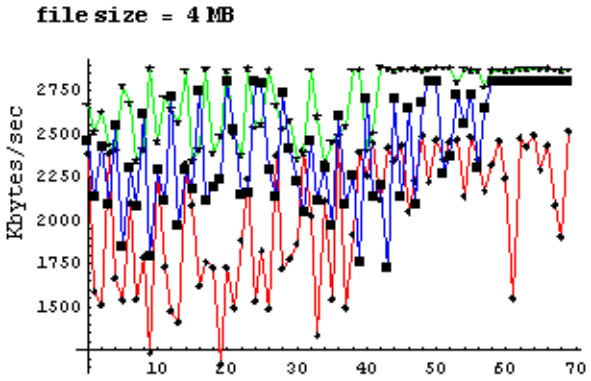


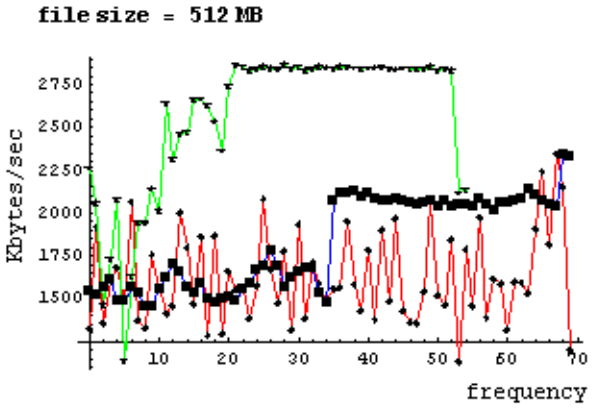
Based on

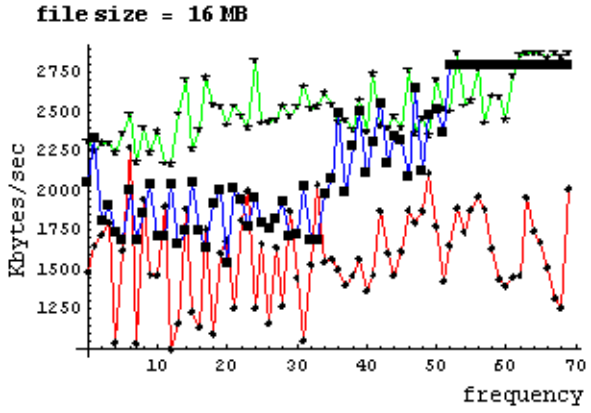
Read performance client1

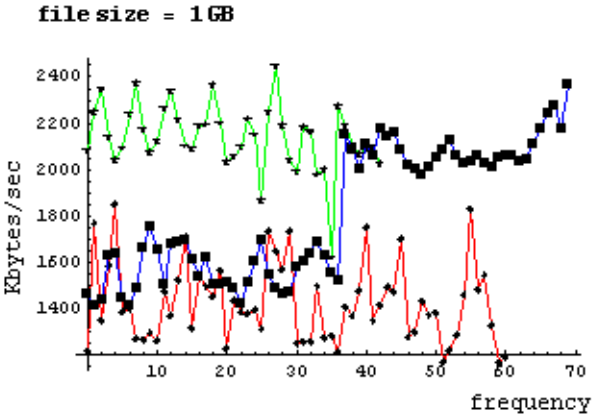




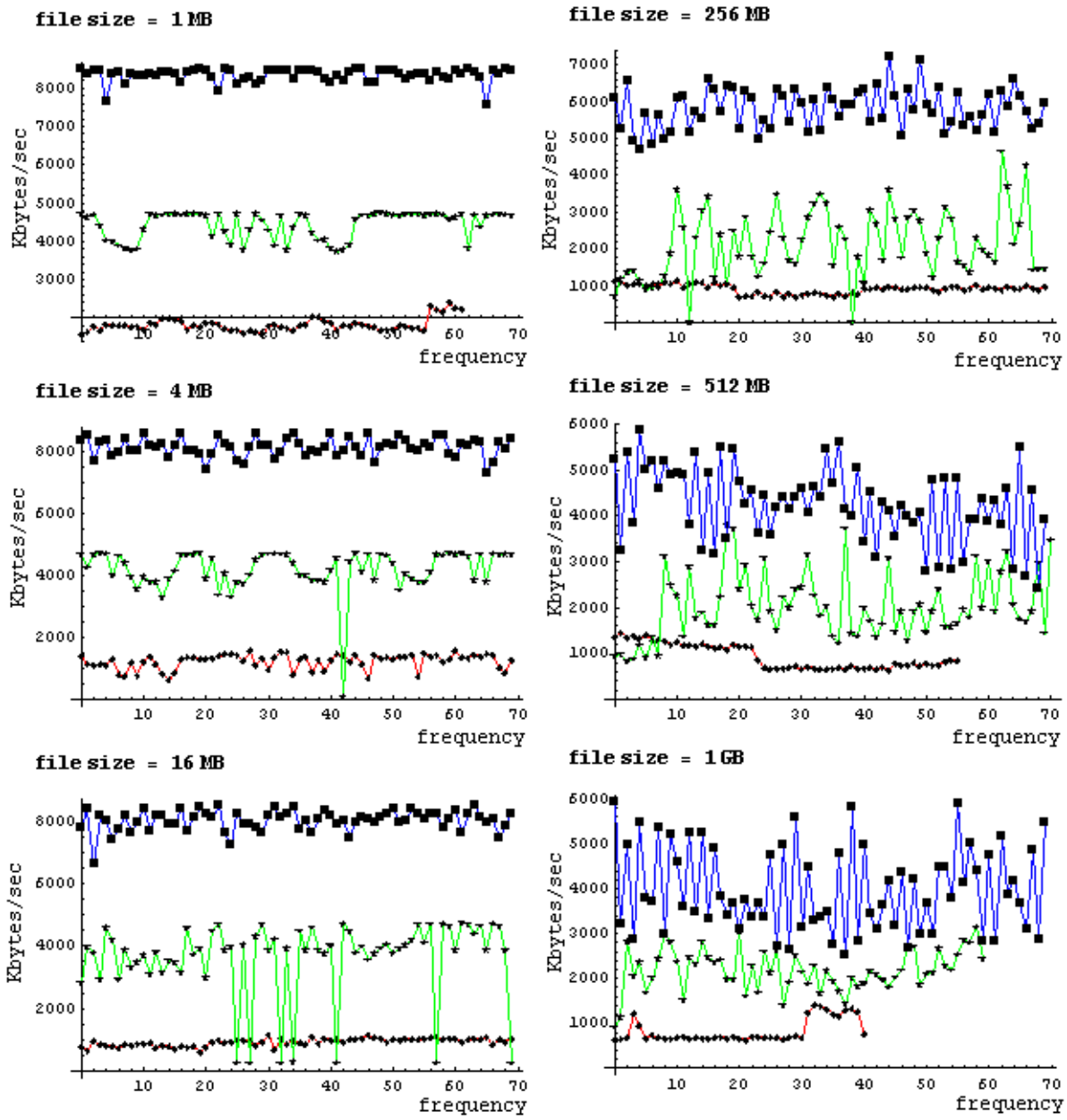






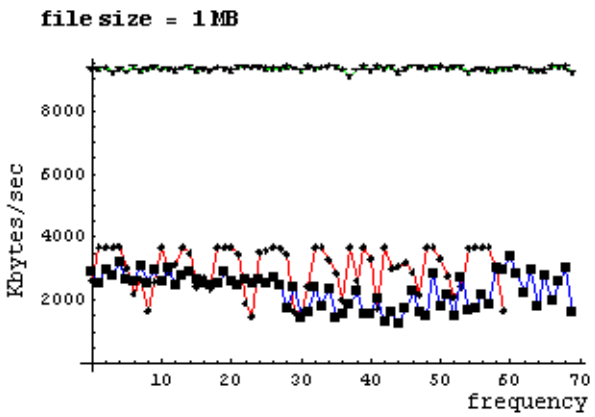


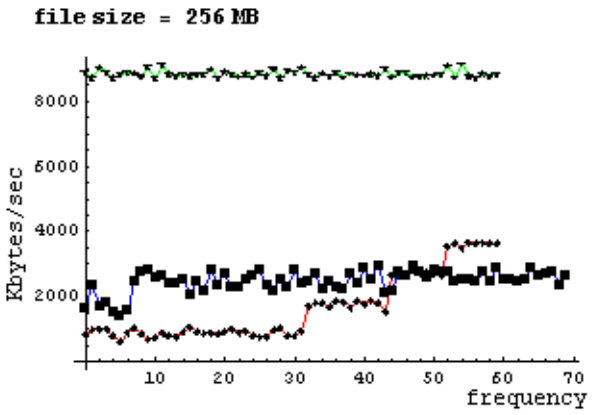
Read performance client2

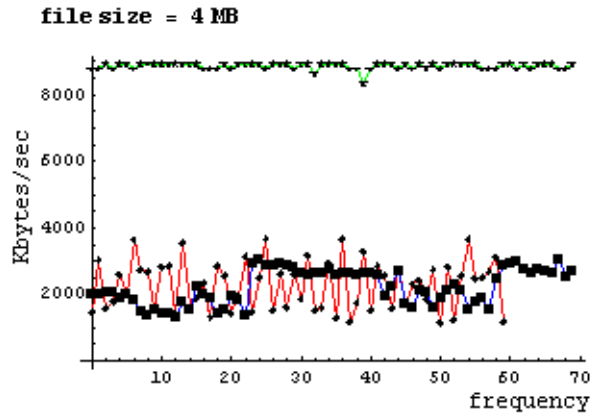


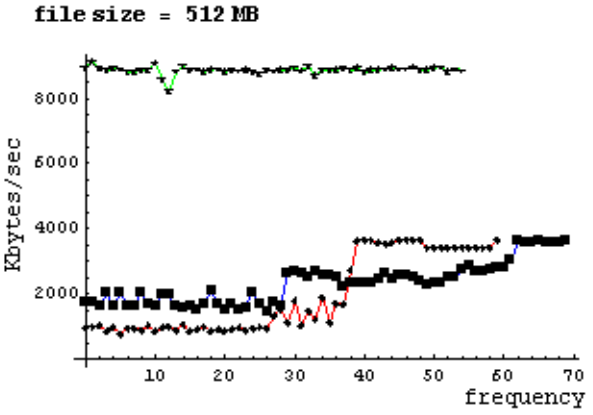
Read performance

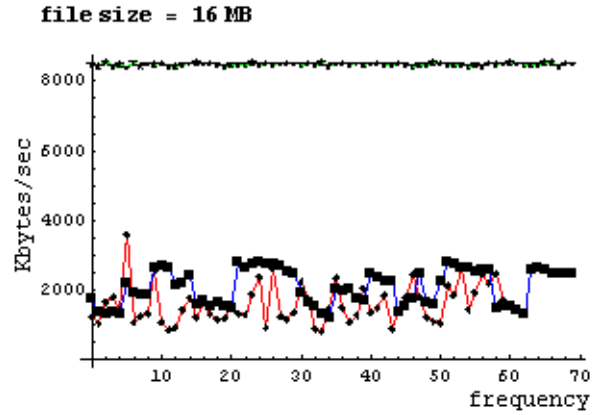
linux client

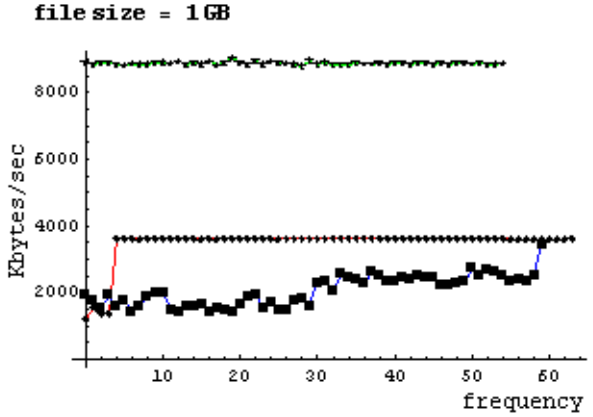






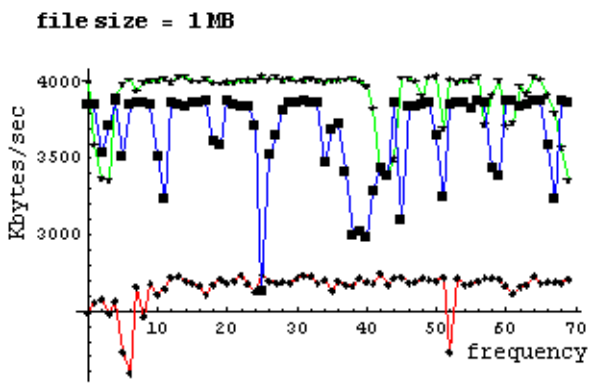


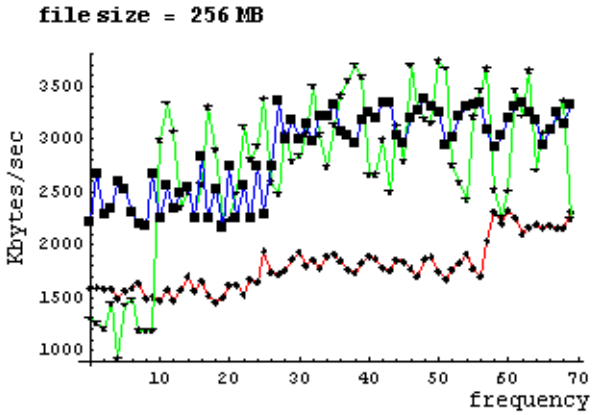


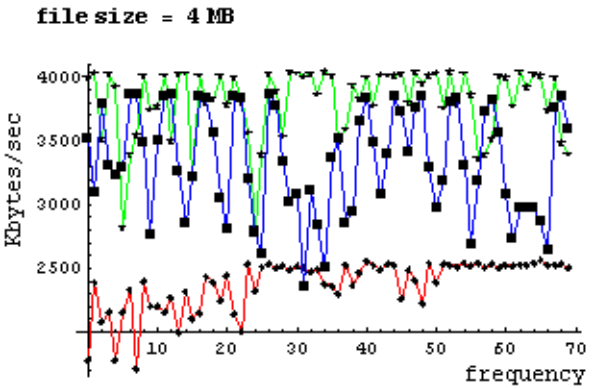


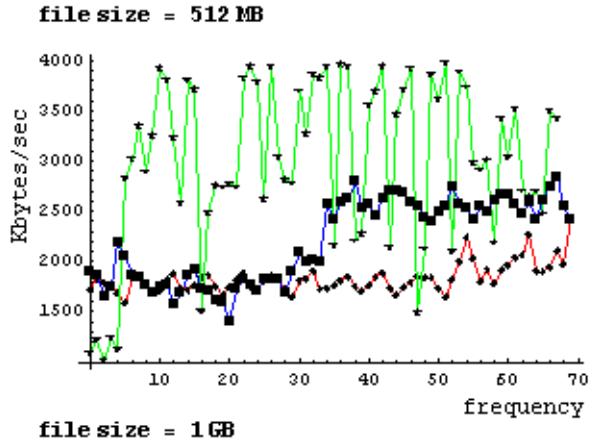
Read performance

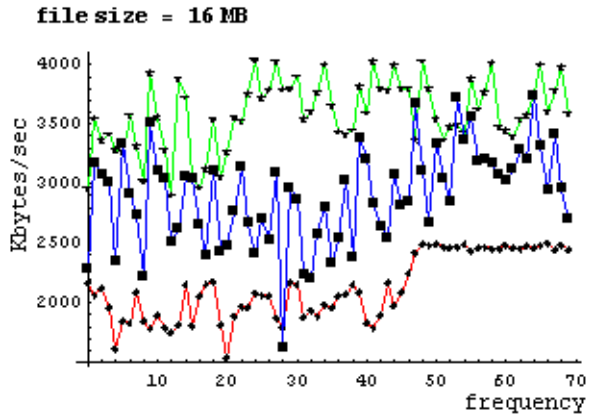
client3

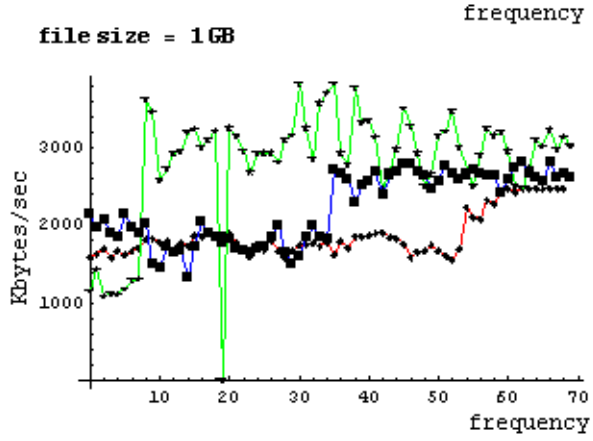








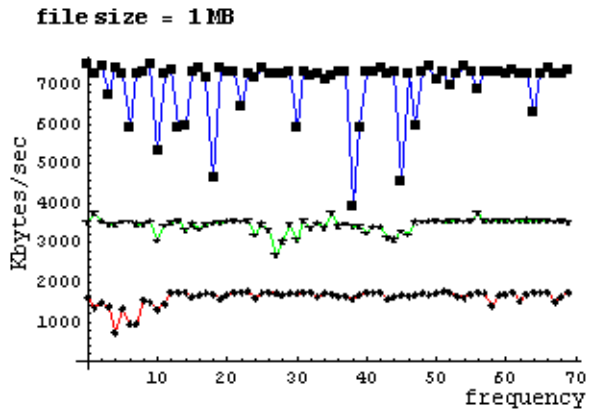


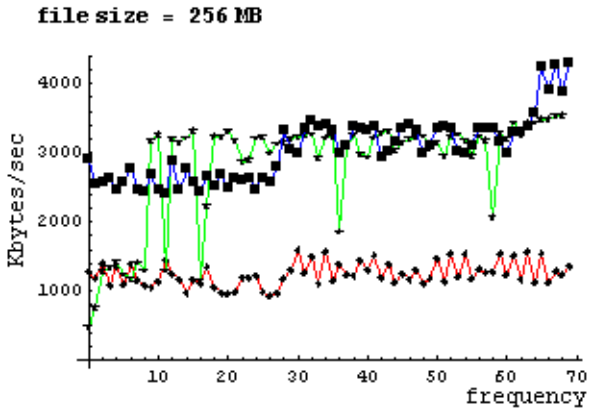


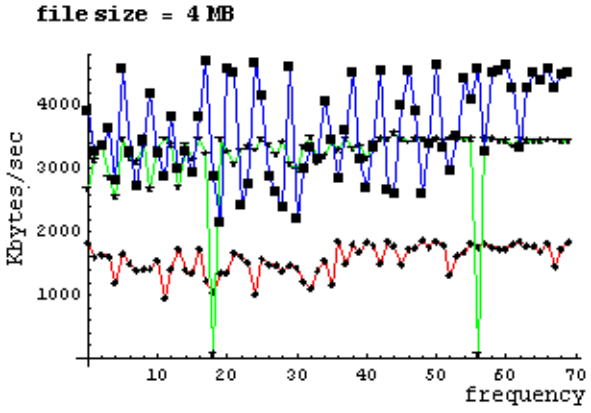
Overlaid lines write performance graphs

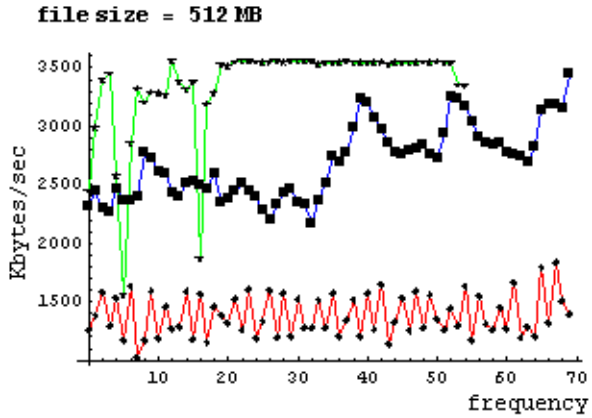
Write performance

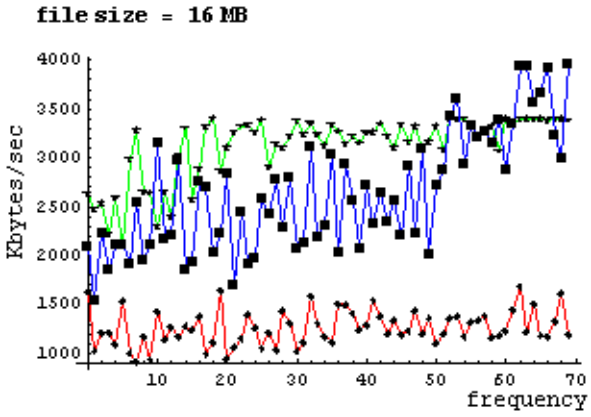
client1

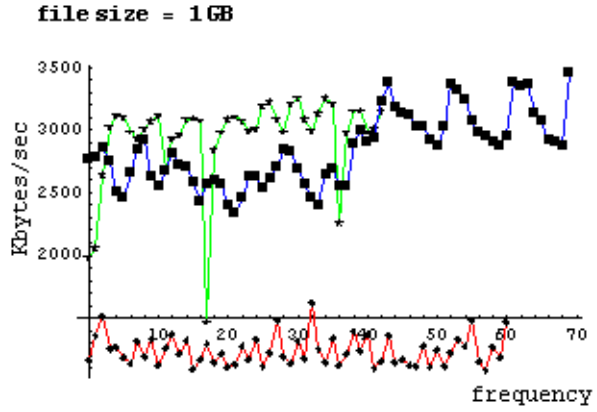






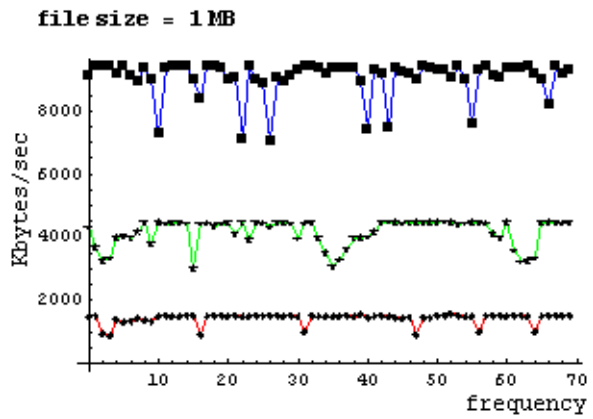


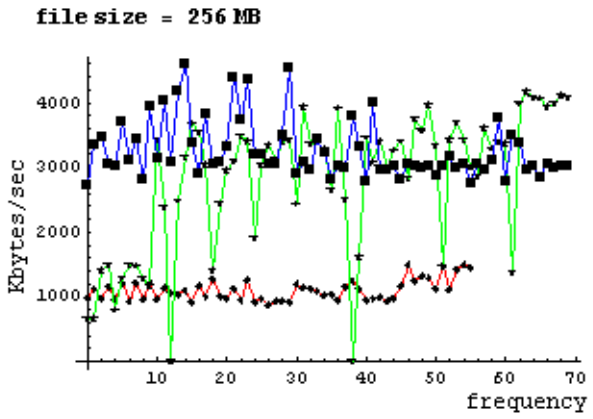


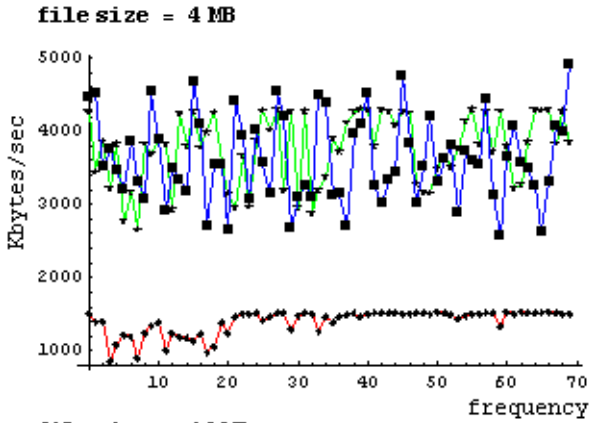


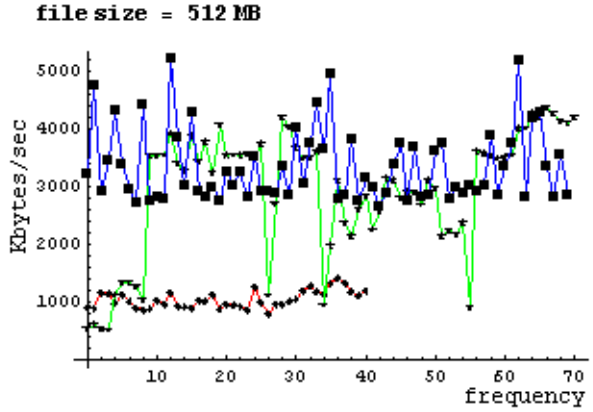
Write performance

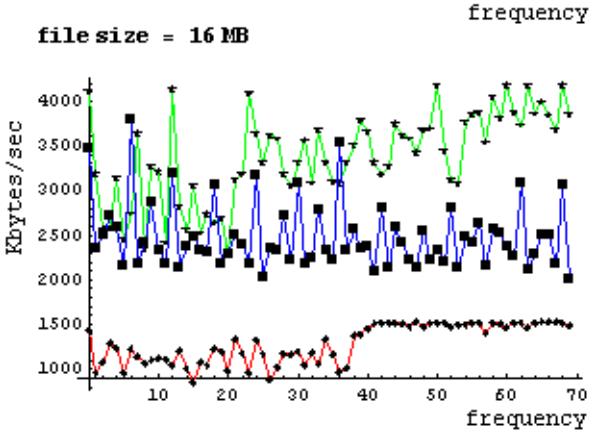
client2

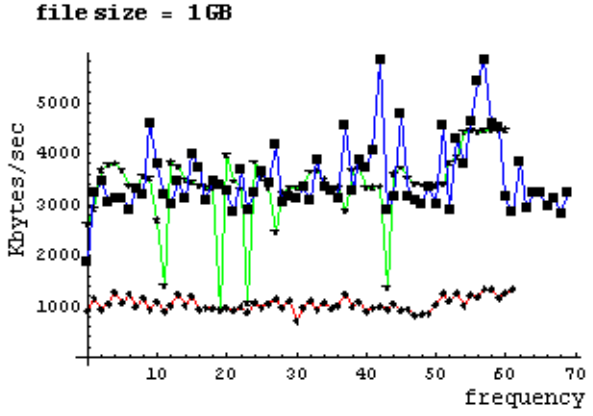




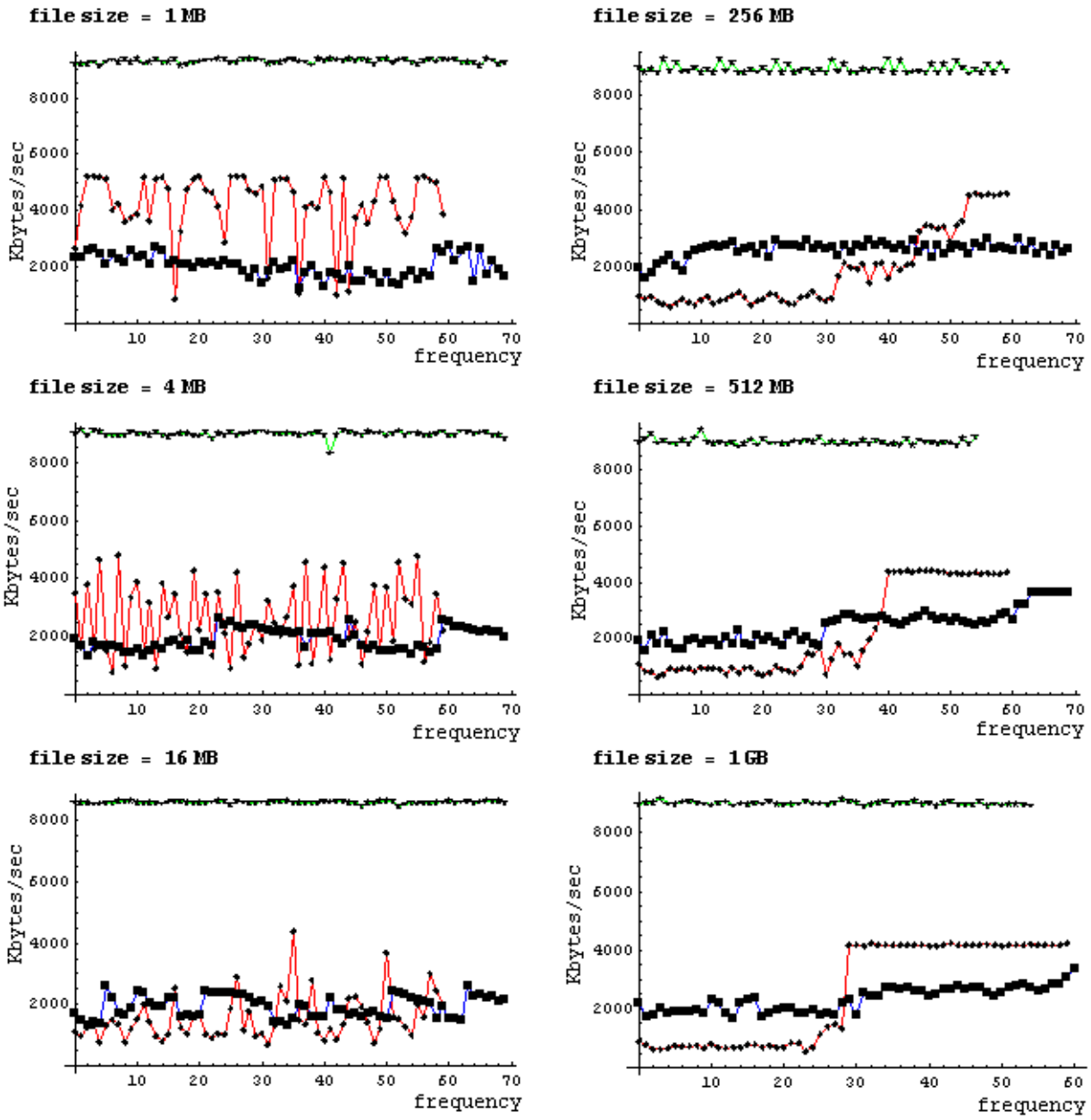






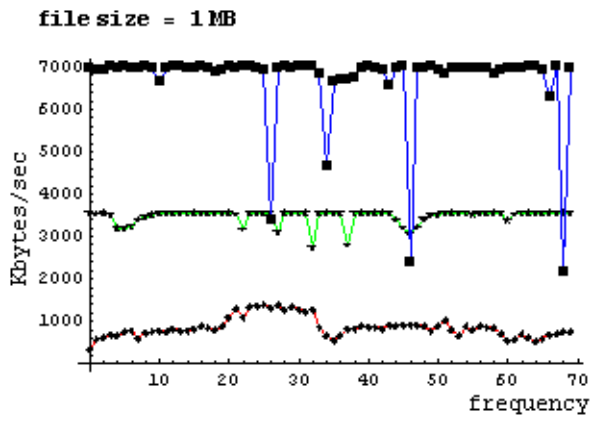


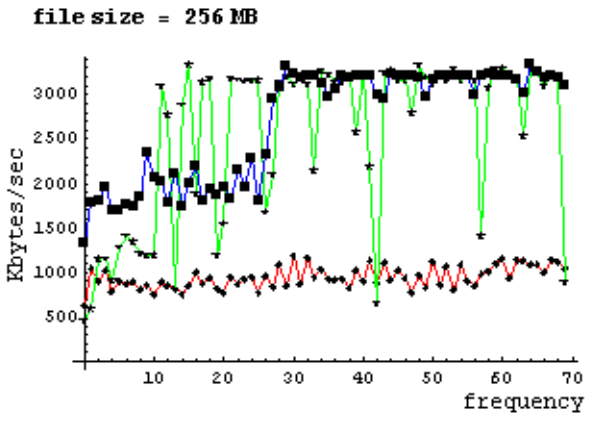
Write performance *linux client*

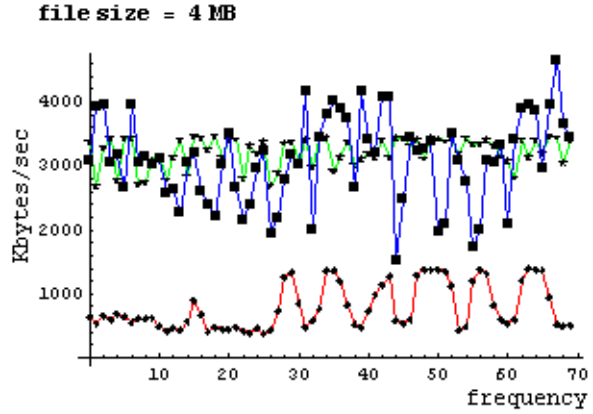


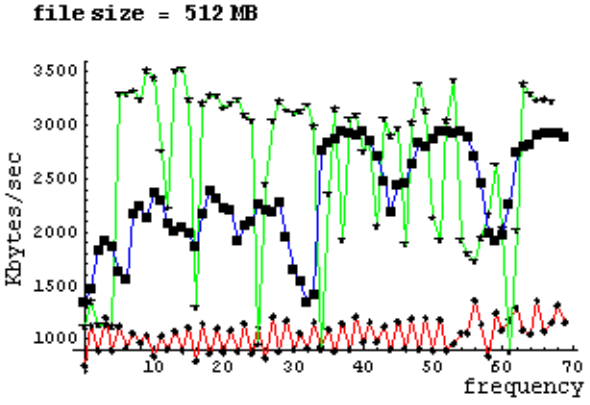
Write performance

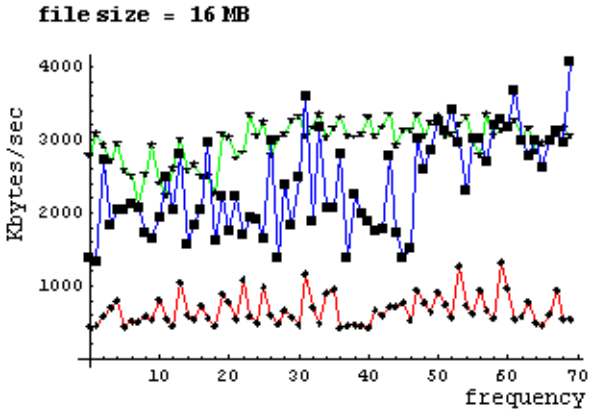
client3

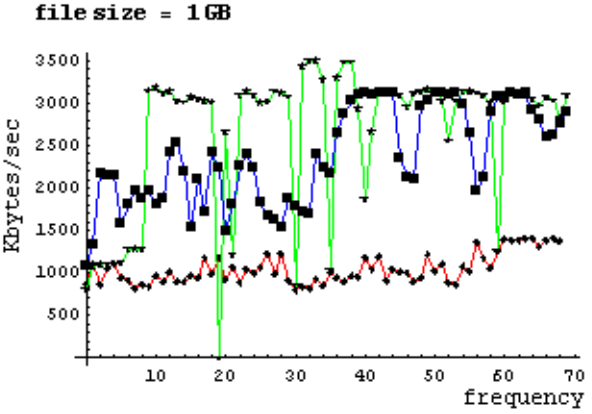








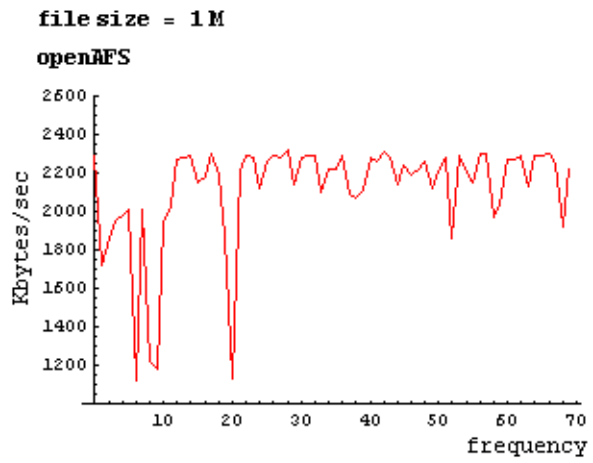


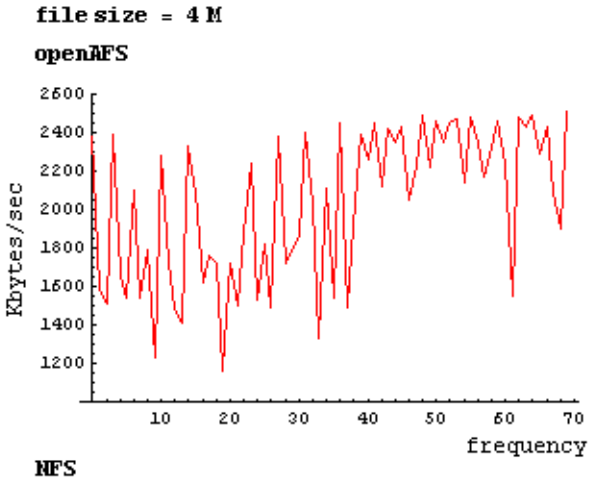


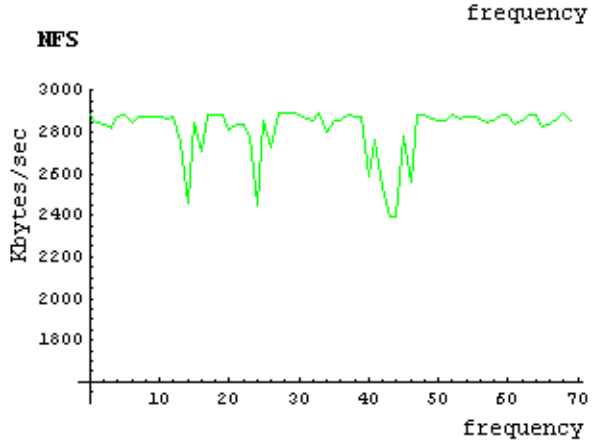
Appendix D

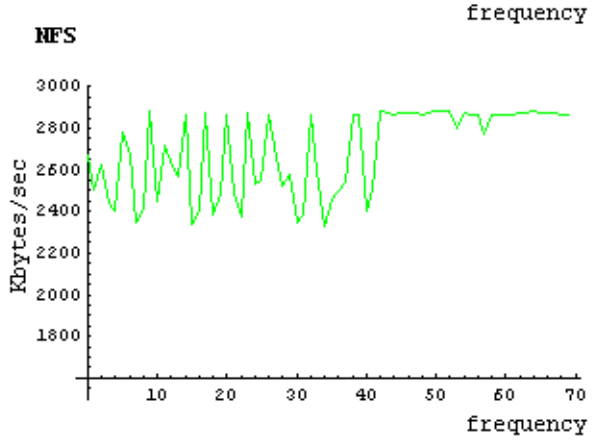
Single line read performance graphs

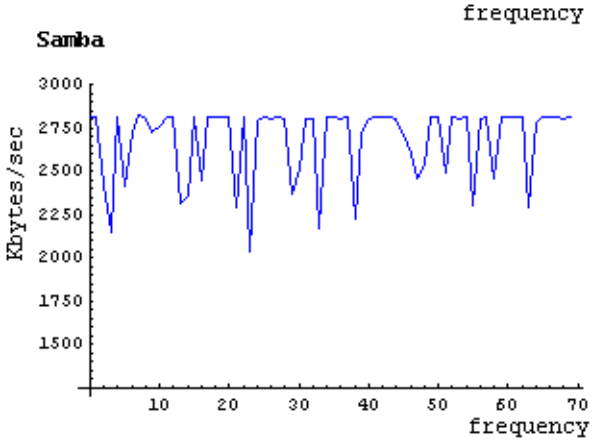
Read performance for client1

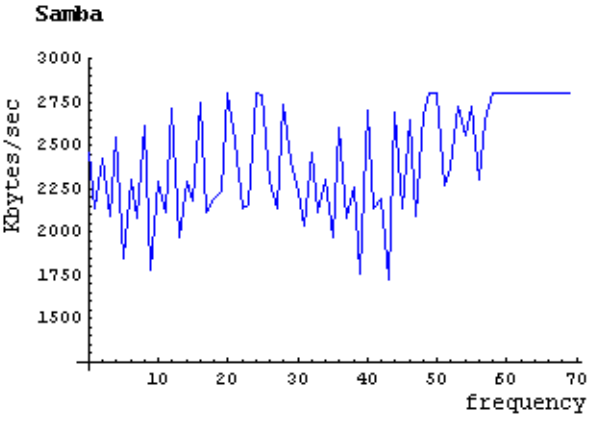


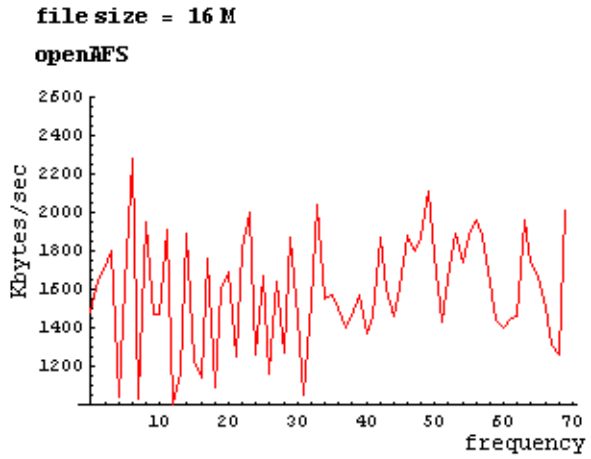


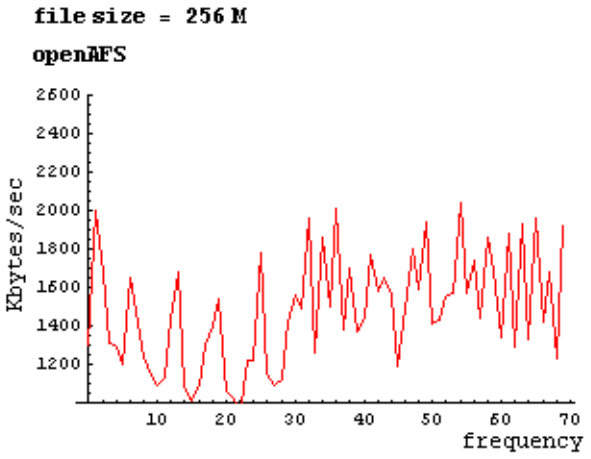


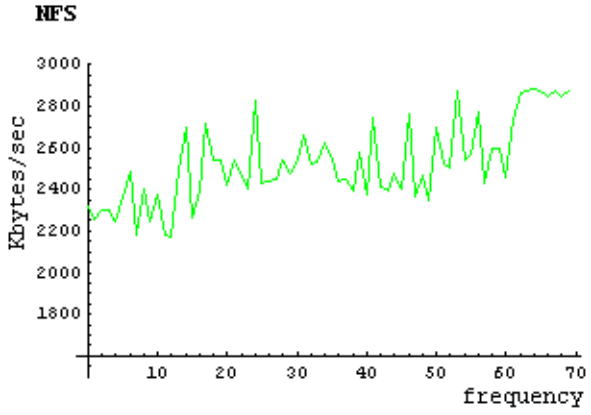


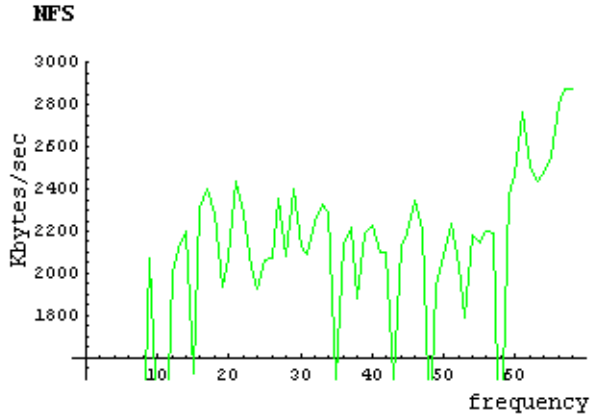


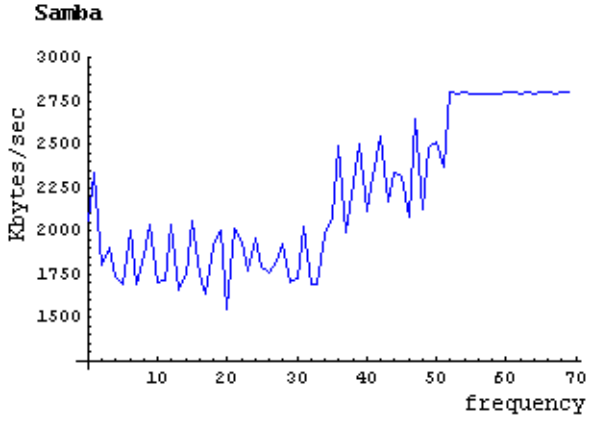


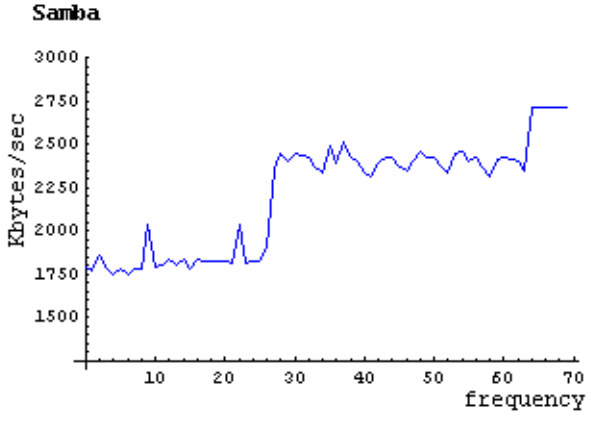

file size = 16 M
openAFS
Kbytes/sec
2600
2400
2200
2000
1800
1600
1400
1200
10
20
30
40
50
60
70
frequency

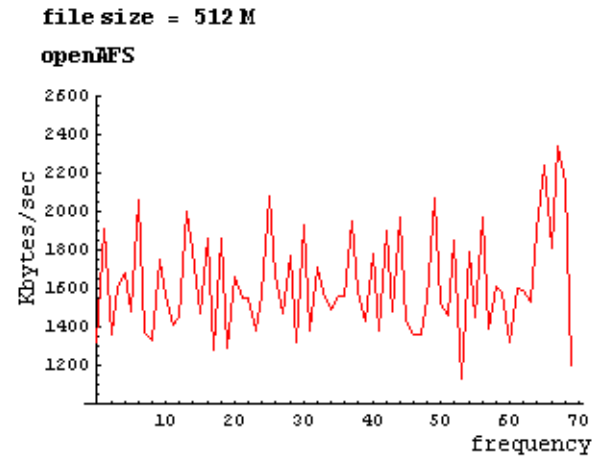

file size = 256 M
openAFS
Kbytes/sec
2600
2400
2200
2000
1800
1600
1400
1200
10
20
30
40
50
60
70
frequency

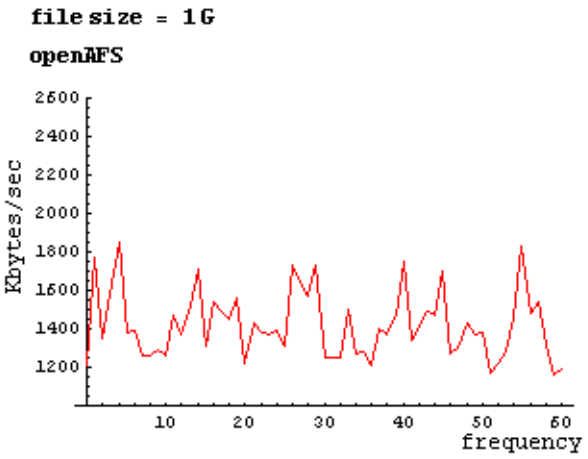

NFS
Kbytes/sec
3000
2800
2600
2400
2200
2000
1800
10
20
30
40
50
60
70
frequency

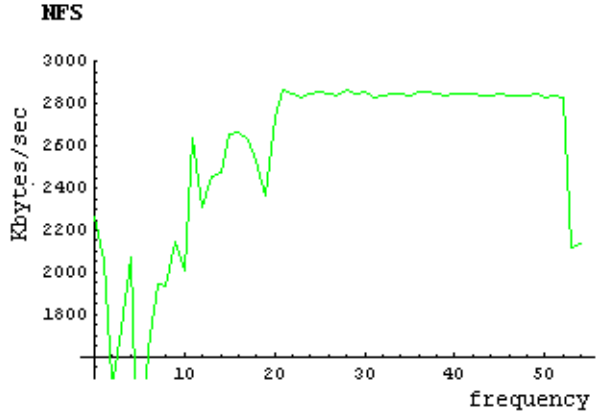

NFS
Kbytes/sec
3000
2800
2600
2400
2200
2000
1800
10
20
30
40
50
60
frequency

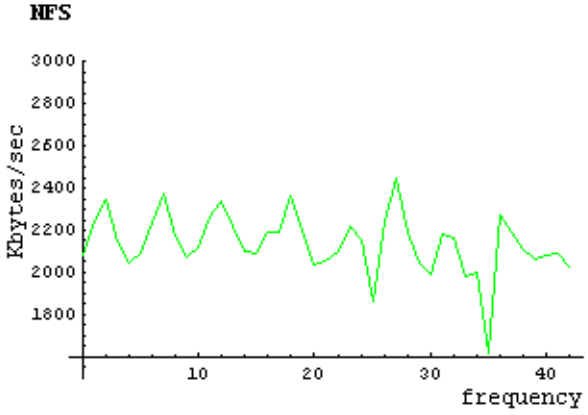

Samba
Kbytes/sec
3000
2750
2500
2250
2000
1750
1500
10
20
30
40
50
60
70
frequency

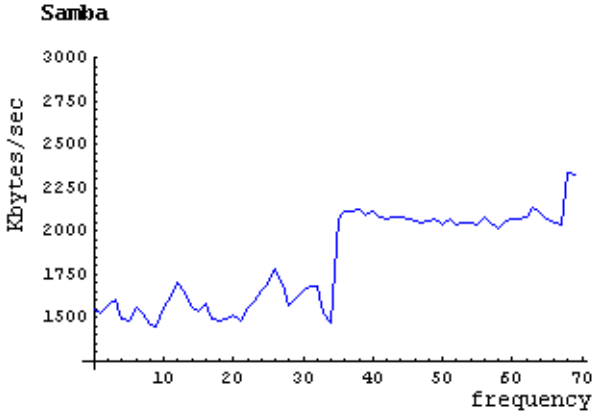

Samba
Kbytes/sec
3000
2750
2500
2250
2000
1750
1500
10
20
30
40
50
60
70
frequency

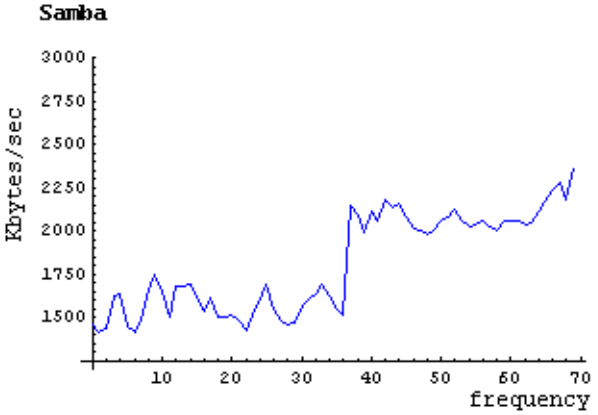

file size = 512 M
openAFS
Kbytes/sec
2600
2400
2200
2000
1800
1600
1400
1200
10
20
30
40
50
60
70
frequency


file size = 1G
openAFS
Kbytes/sec
2600
2400
2200
2000
1800
1600
1400
1200
10
20
30
40
50
60
frequency

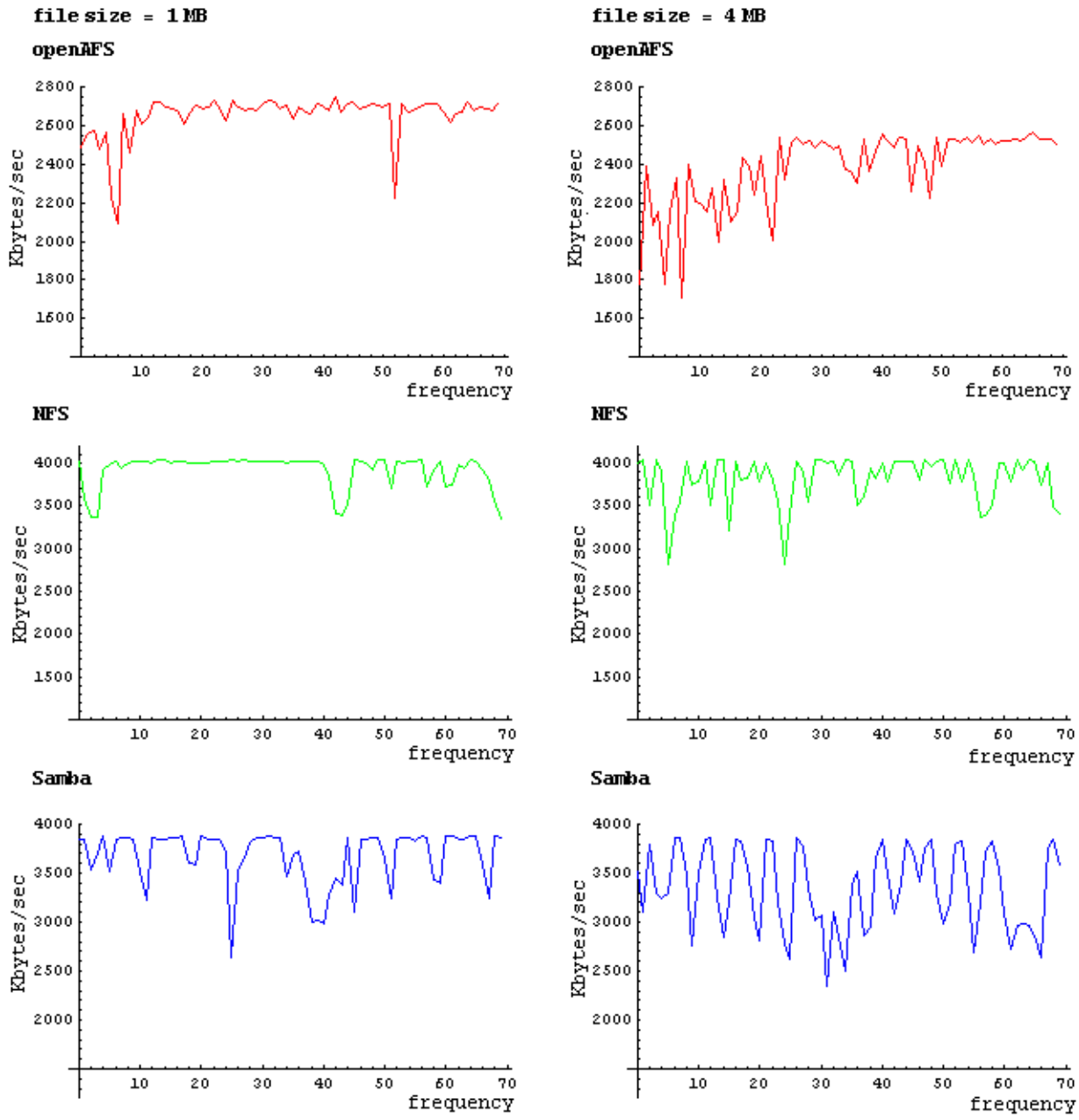

NFS
Kbytes/sec
3000
2800
2600
2400
2200
2000
1800
10
20
30
40
50
frequency

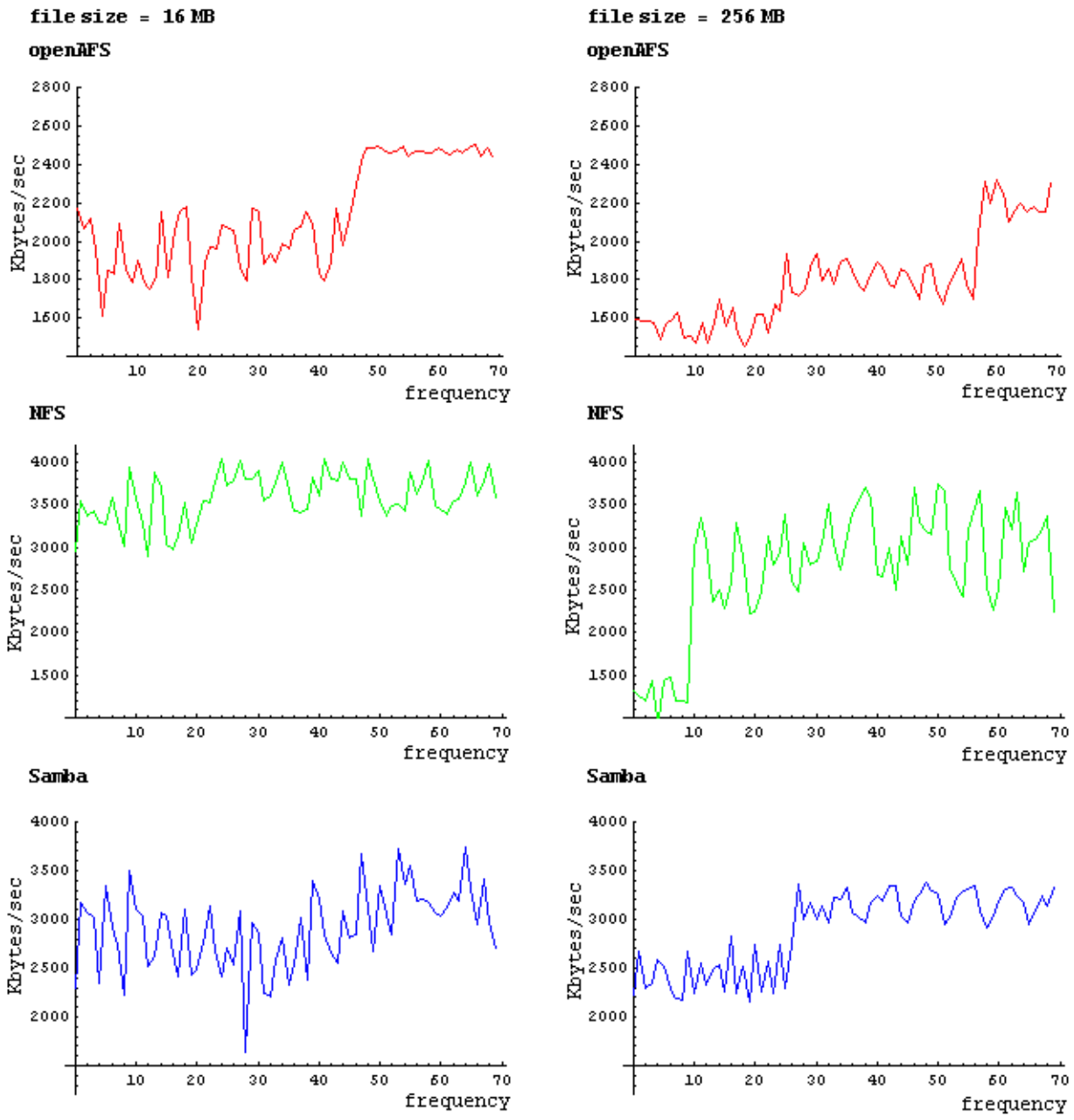

NFS
Kbytes/sec
3000
2800
2600
2400
2200
2000
1800
10
20
30
40
frequency

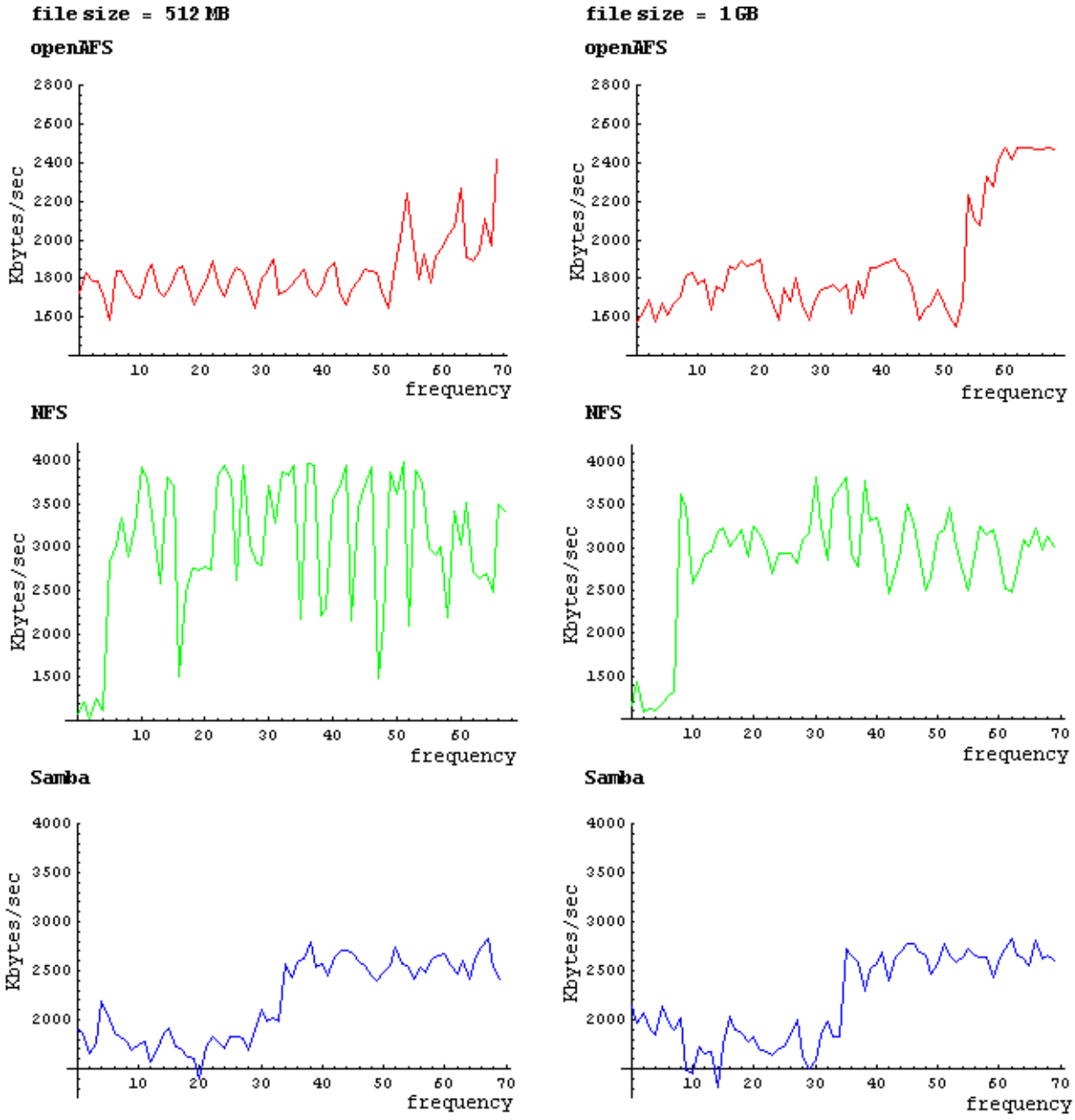

Samba
Kbytes/sec
3000
2750
2500
2250
2000
1750
1500
10
20
30
40
50
60
70
frequency


Samba
Kbytes/sec
3000
2750
2500
2250
2000
1750
1500
10
20
30
40
50
60
70
frequency

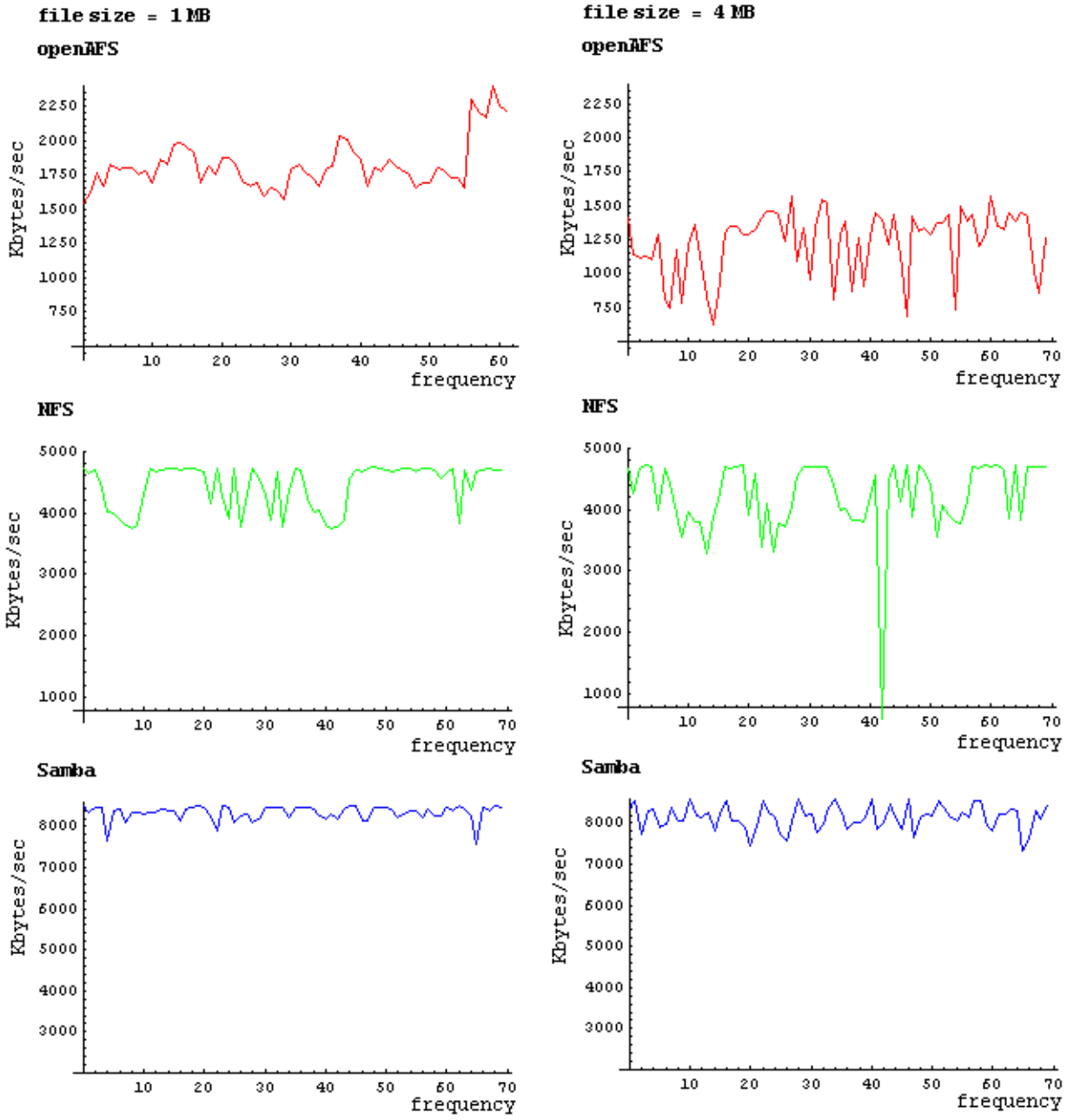
Read performance for client3

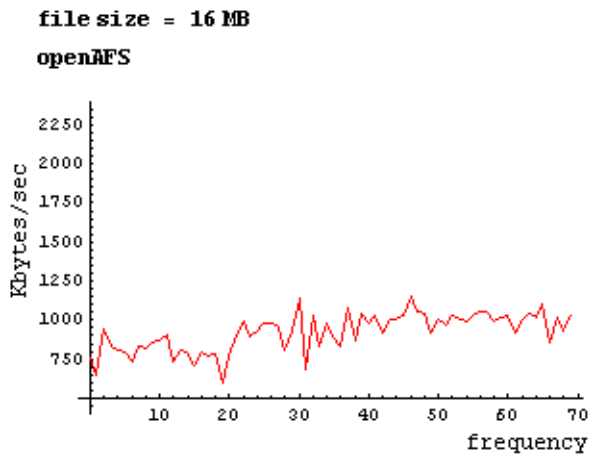


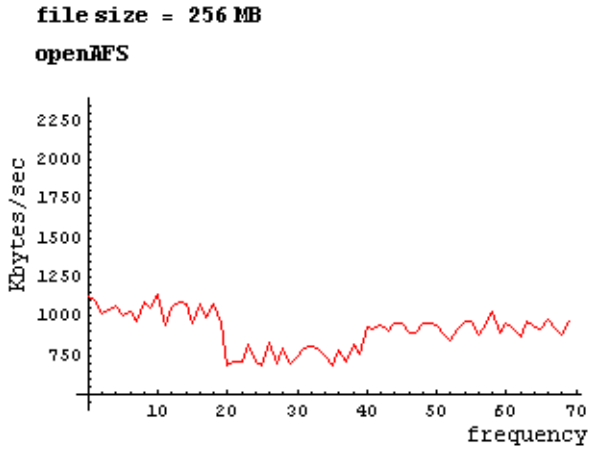

file size = 16 MB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency
file size = 256 MB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency

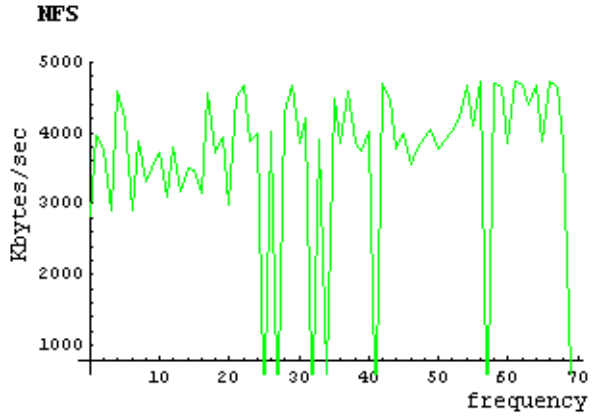

file size = 512 MB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency
file size = 1 GB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency

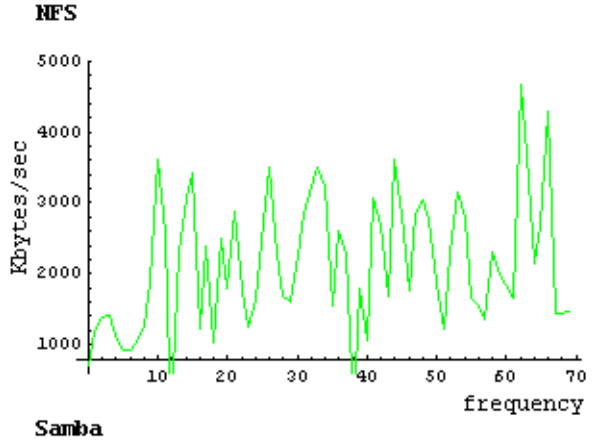
Read performance for client2

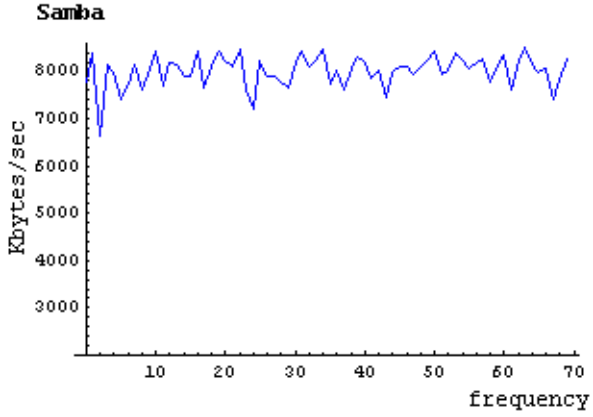


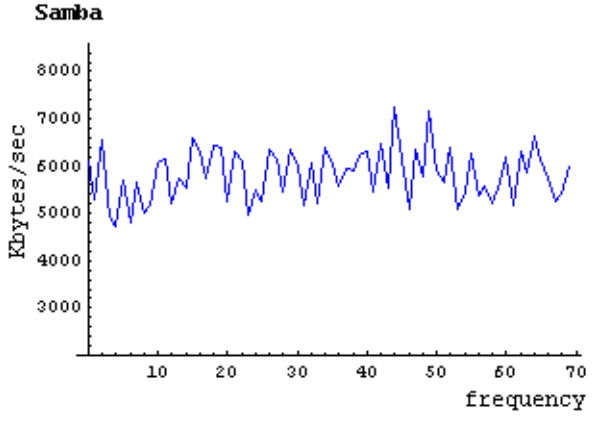

file size = 16 MB
openAFS
Kbytes/sec
2250
2000
1750
1500
1250
1000
750
10
20
30
40
50
60
70
frequency

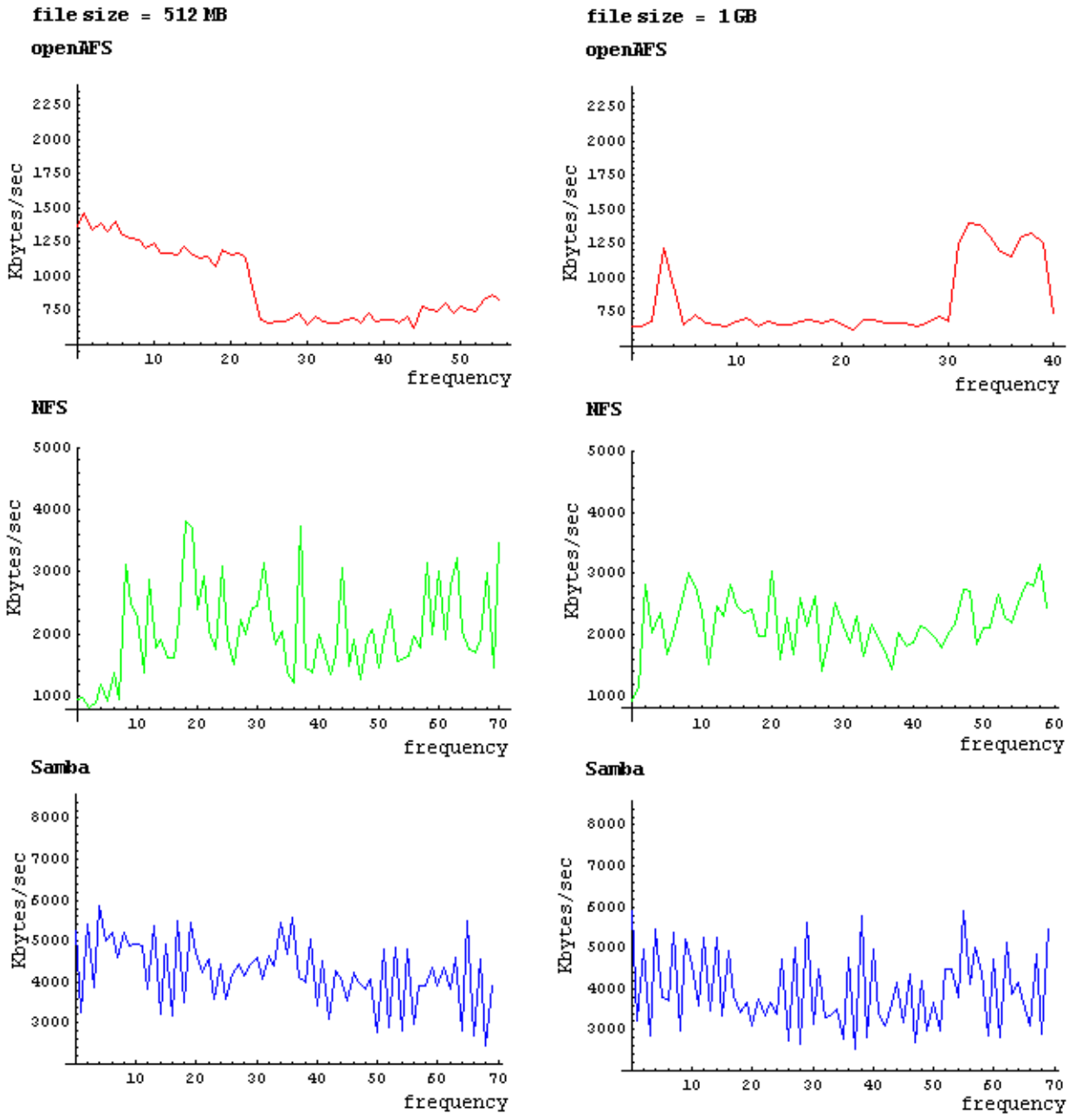

file size = 256 MB
openAFS
Kbytes/sec
2250
2000
1750
1500
1250
1000
750
10
20
30
40
50
60
70
frequency


NFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
70
frequency

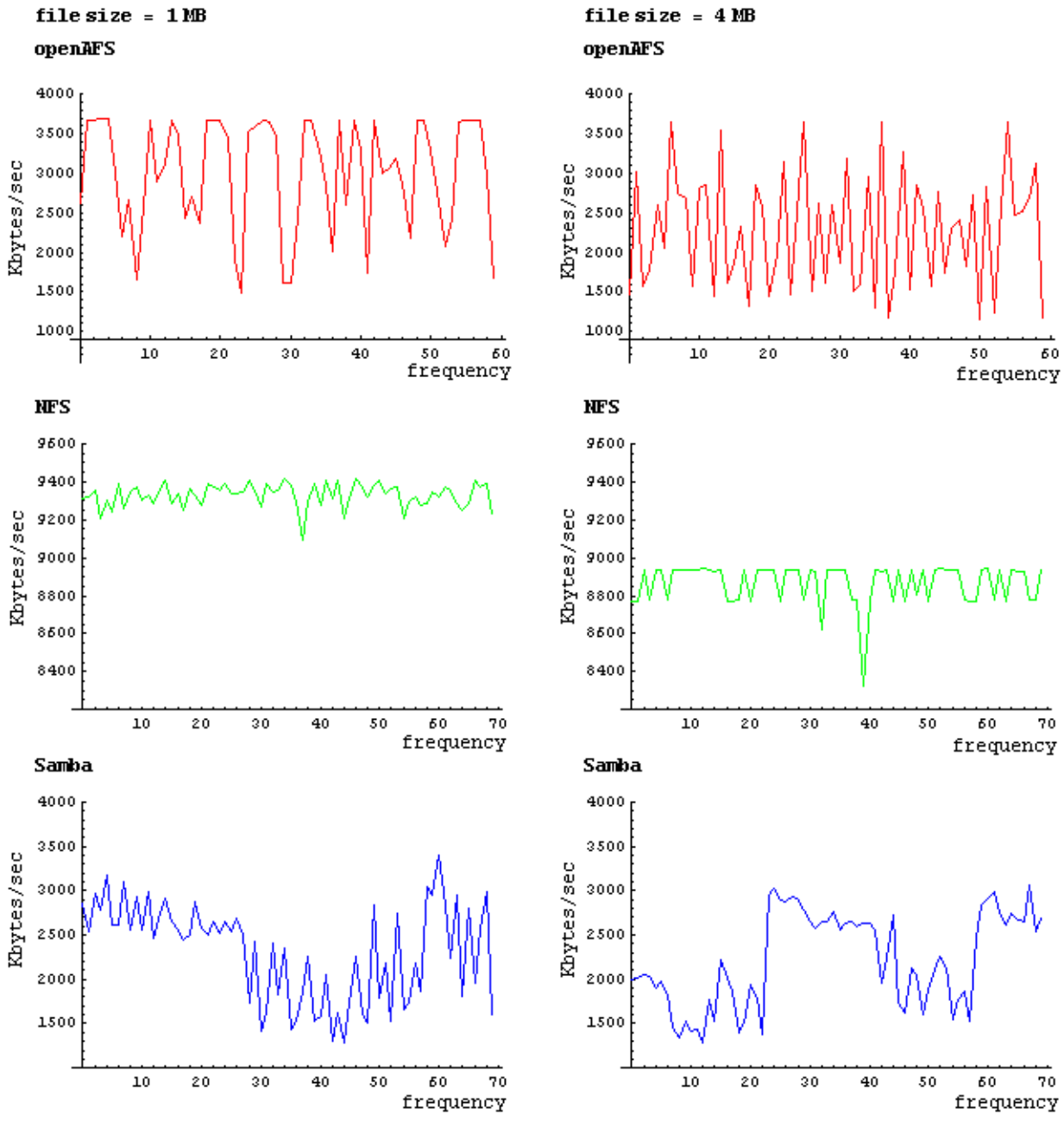

NFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
70
frequency

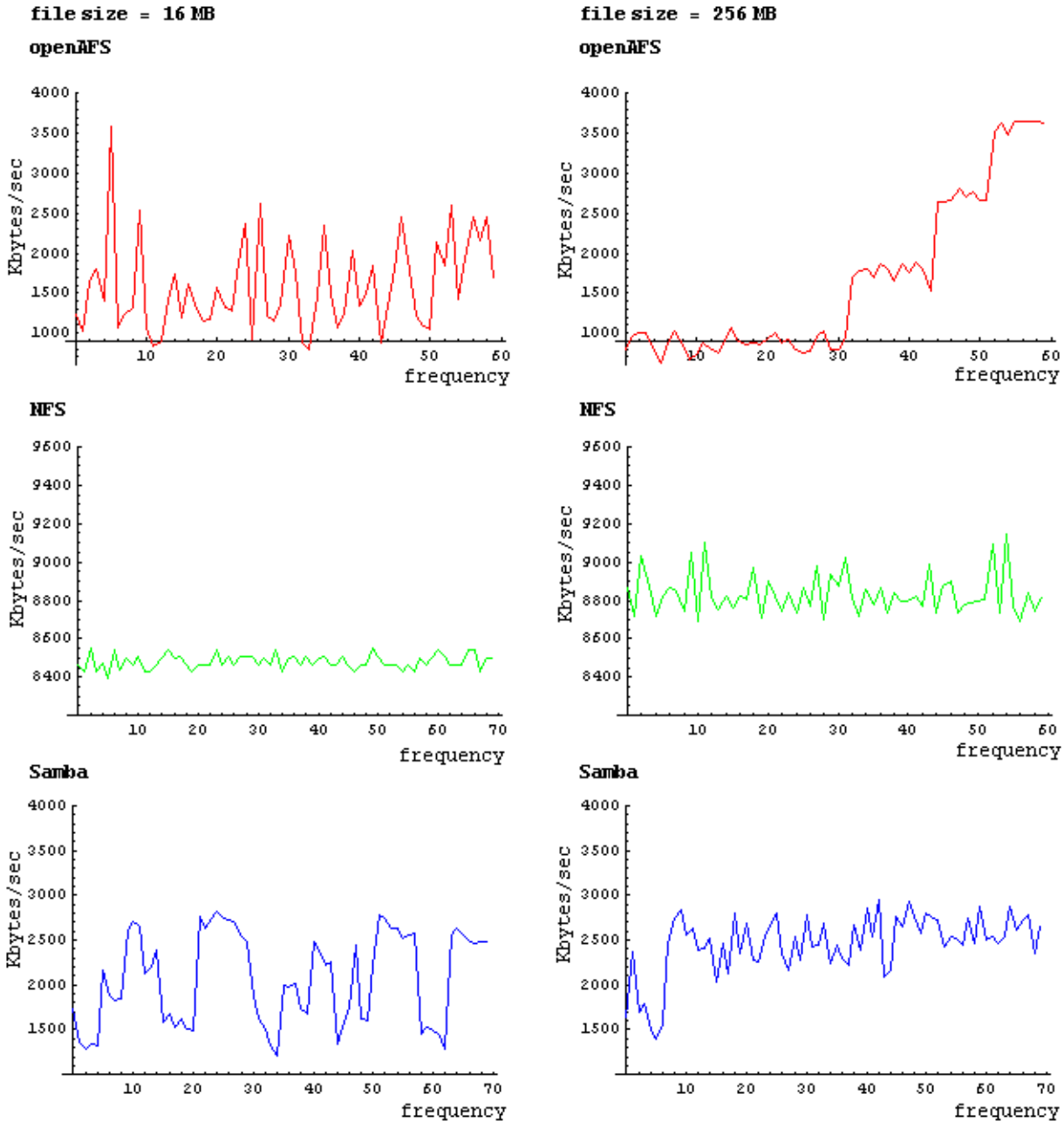

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

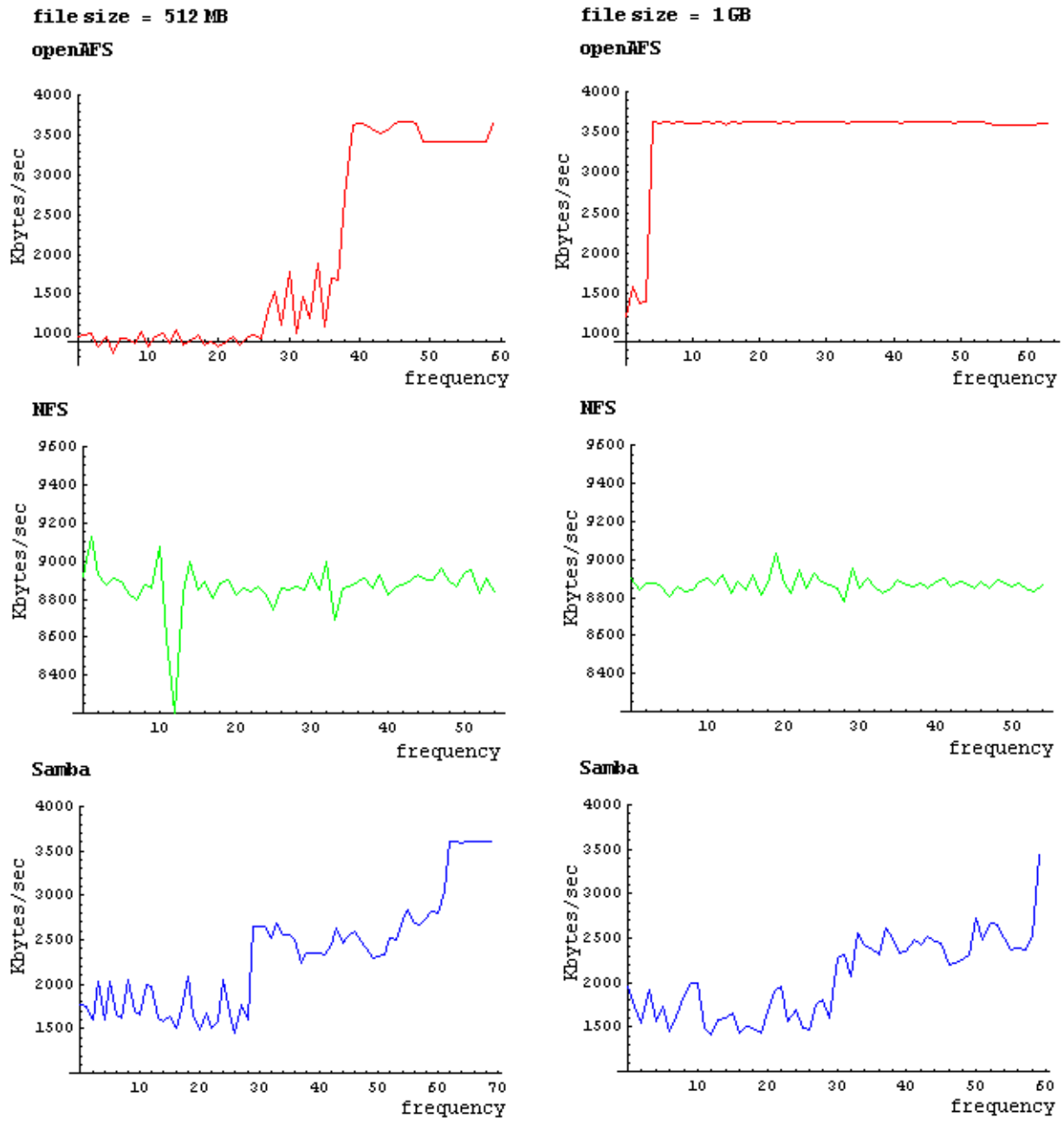

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency


file size = 512 MB
openAFS
Kbytes/sec
frequency
NFS
Samba
file size = 1 GB
openAFS
Kbytes/sec
frequency
NFS
Samba

Read performance for *linux client*

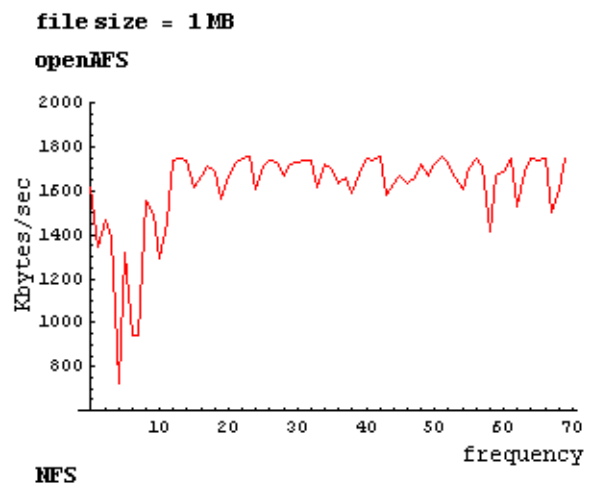


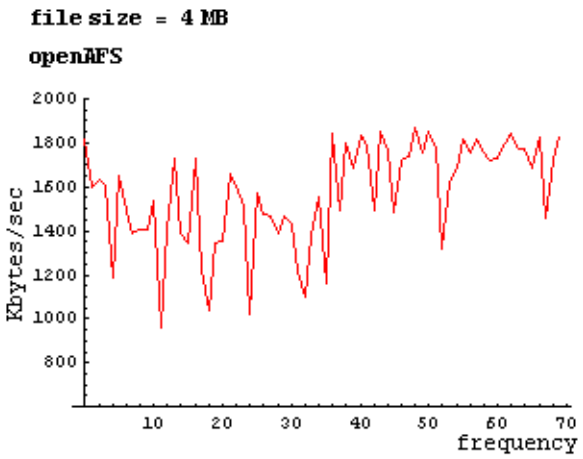

file size = 16 MB
openAFS
Kbytes/sec
frequency
file size = 256 MB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency

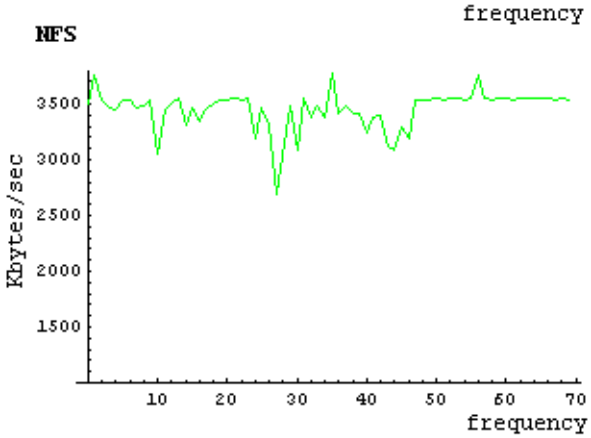

file size = 512 MB
openAFS
Kbytes/sec
frequency
file size = 1GB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency

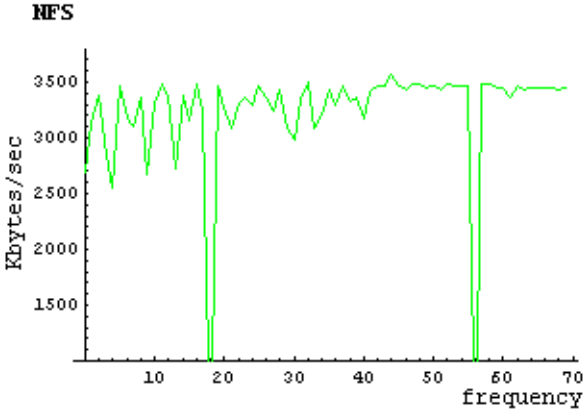
Single line read performance graphs

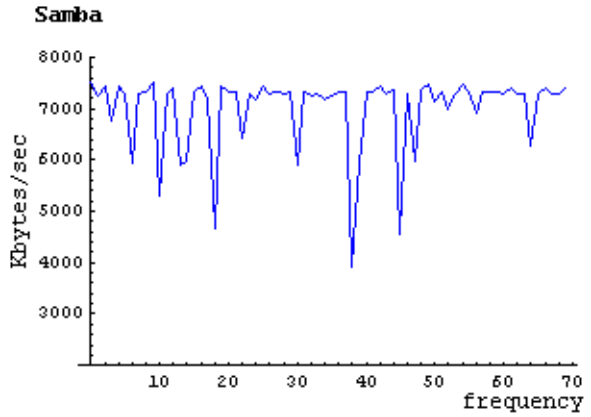
Write performance for client1

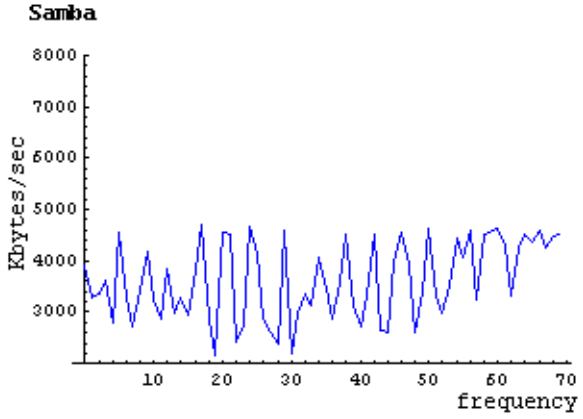


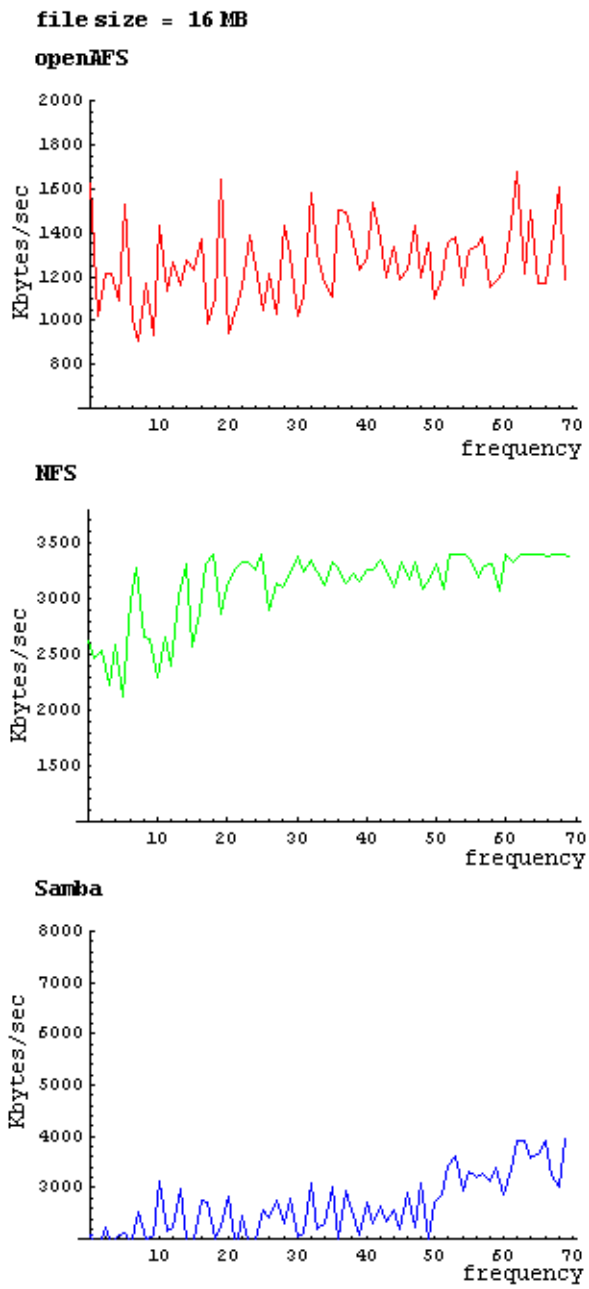


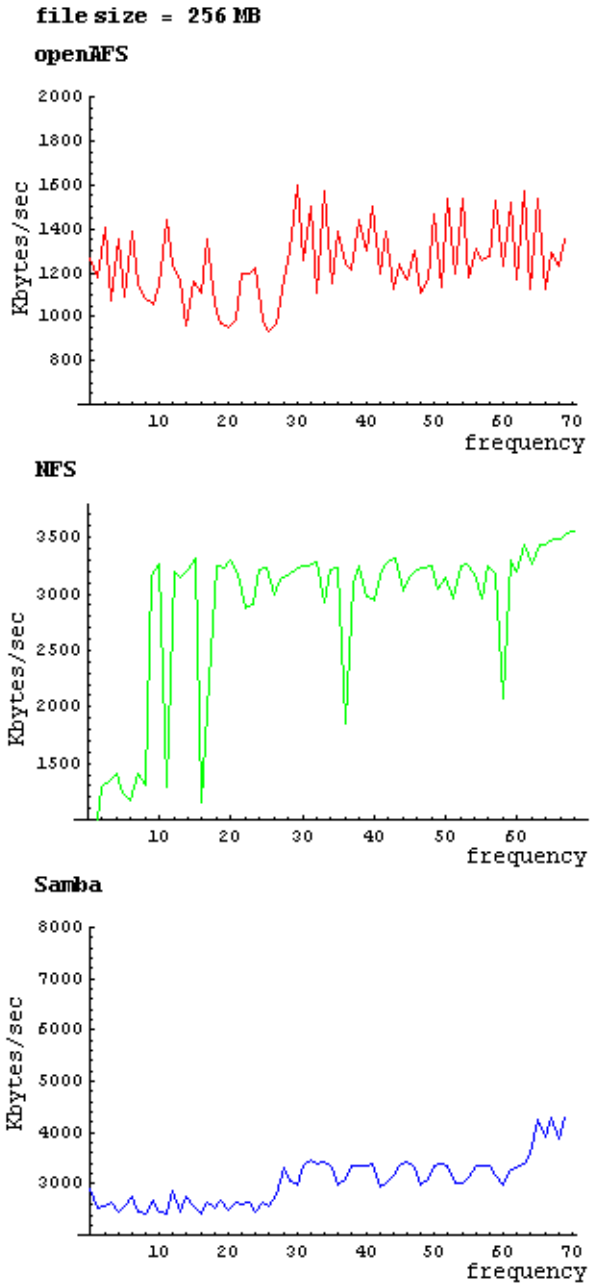


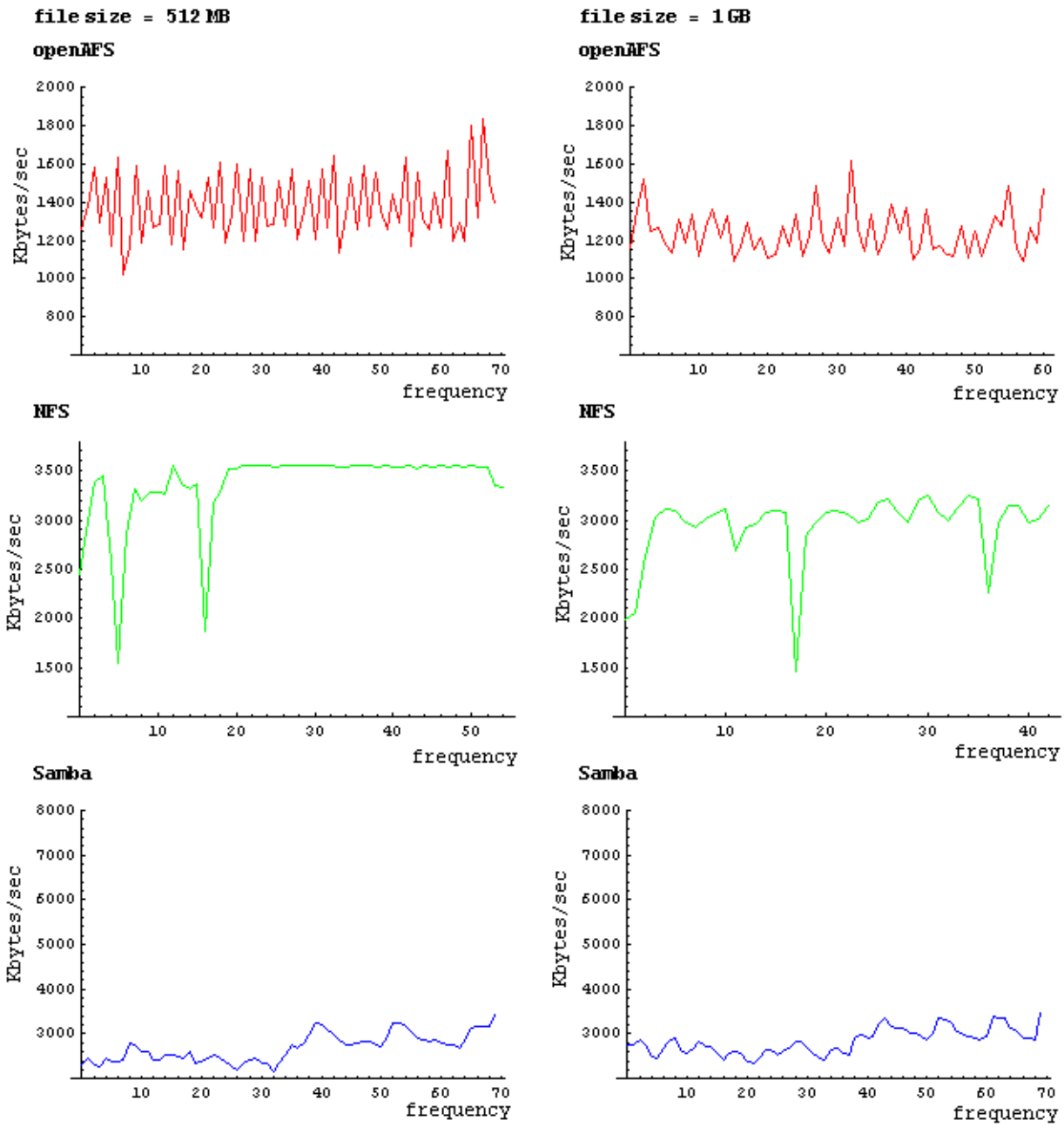




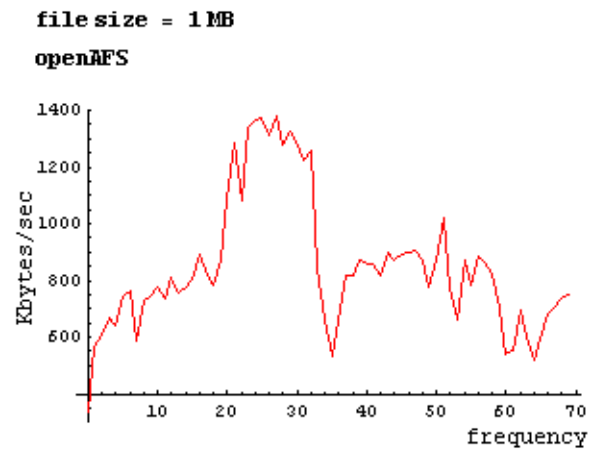


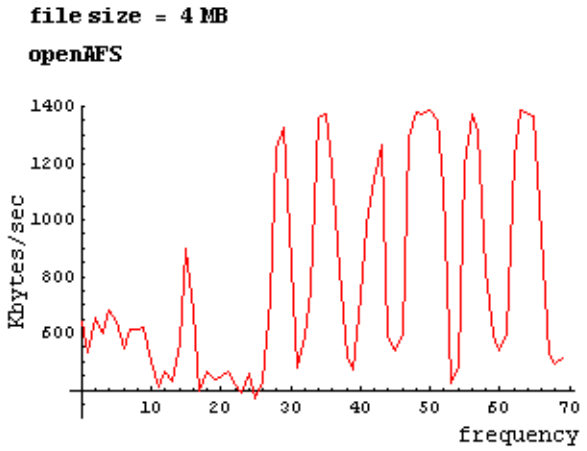

file size = 16 MB
openAFS
Kbytes/sec
2000
1800
1600
1400
1200
1000
800
10
20
30
40
50
60
70
frequency
NFS
Kbytes/sec
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency
Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

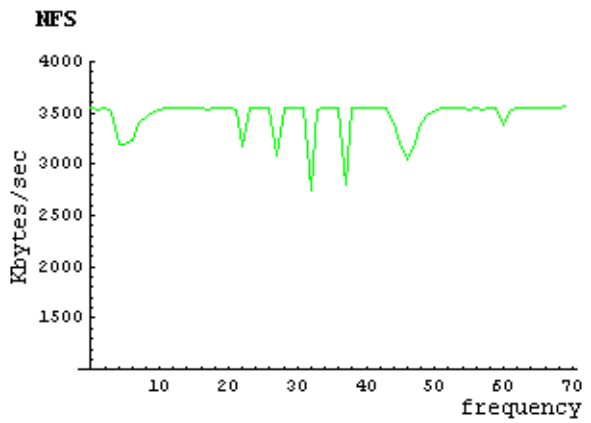

file size = 256 MB
openAFS
Kbytes/sec
2000
1800
1600
1400
1200
1000
800
10
20
30
40
50
60
70
frequency
NFS
Kbytes/sec
3500
3000
2500
2000
1500
10
20
30
40
50
60
frequency
Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

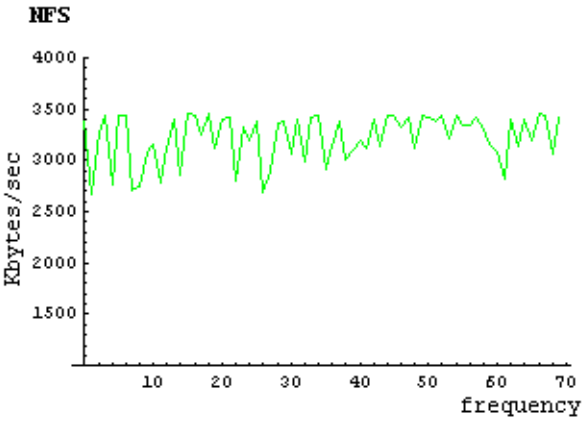

file size = 512 MB
openAFS
Kbytes/sec
2000
1800
1600
1400
1200
1000
800
10
20
30
40
50
60
70
frequency
NFS
Kbytes/sec
3500
3000
2500
2000
1500
10
20
30
40
50
frequency
Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency
file size = 1 GB
openAFS
Kbytes/sec
2000
1800
1600
1400
1200
1000
800
10
20
30
40
50
60
frequency
NFS
Kbytes/sec
3500
3000
2500
2000
1500
10
20
30
40
frequency
Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

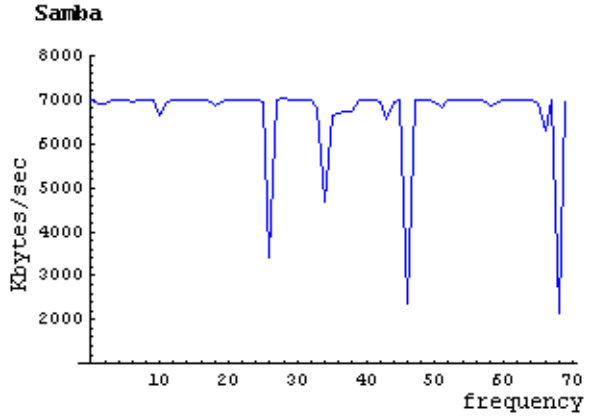
Write performance for client3

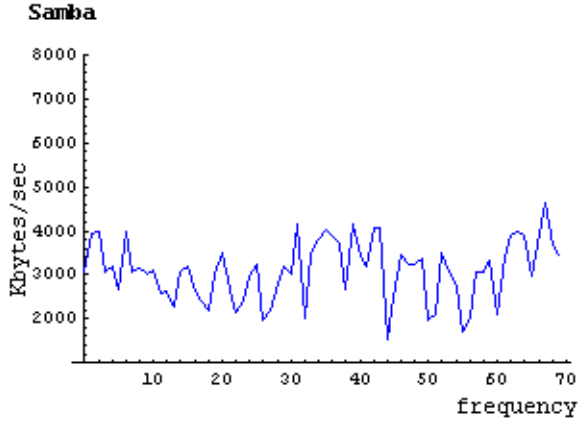


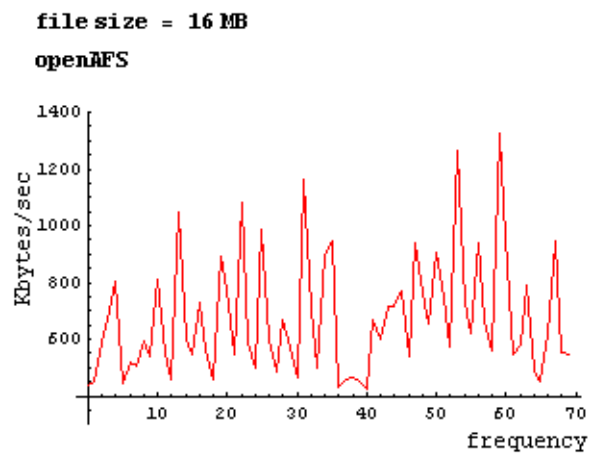


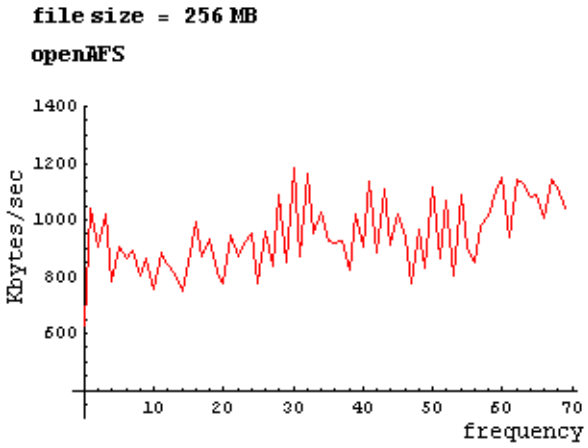


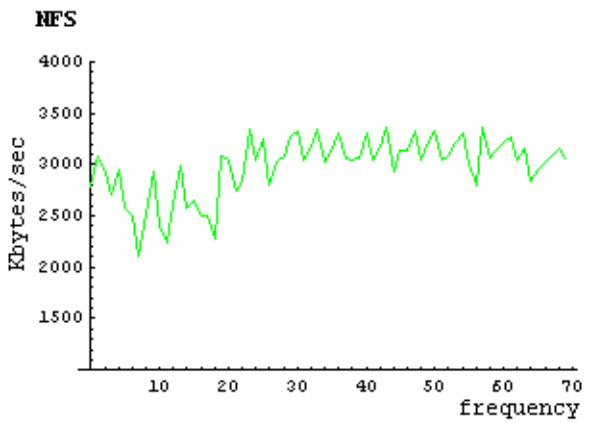


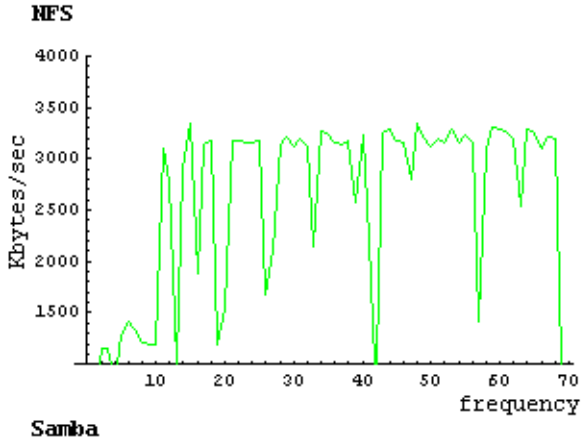


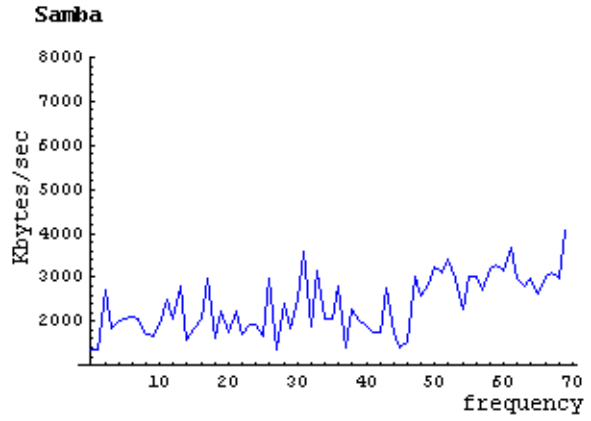


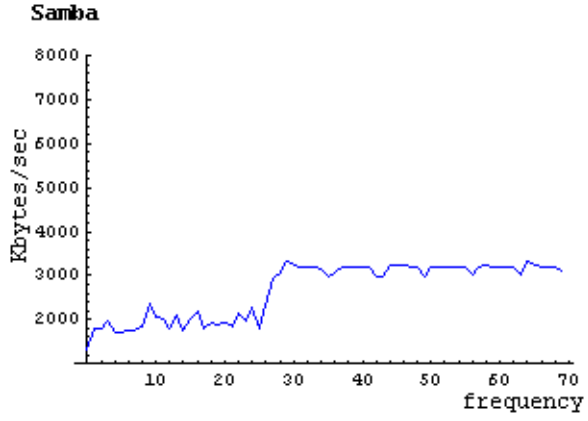

file size = 16 MB
openAFS
Kbytes/sec
1400
1200
1000
800
600
10
20
30
40
50
60
70
frequency

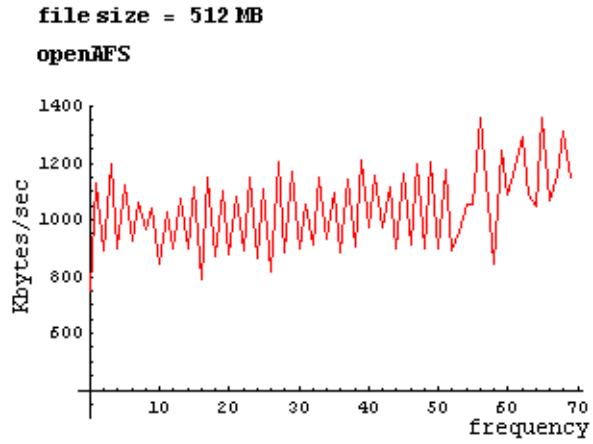

file size = 256 MB
openAFS
Kbytes/sec
1400
1200
1000
800
600
10
20
30
40
50
60
70
frequency

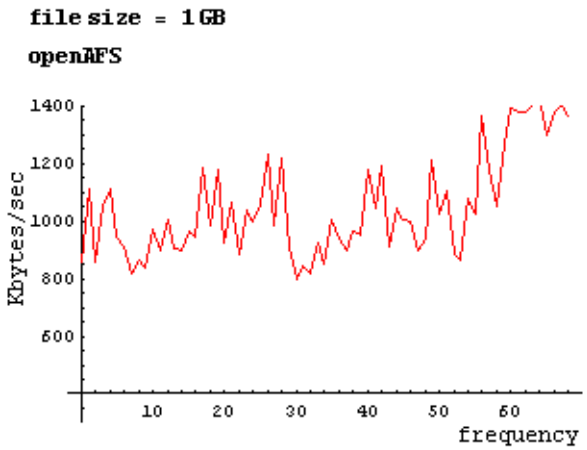

NFS
Kbytes/sec
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency

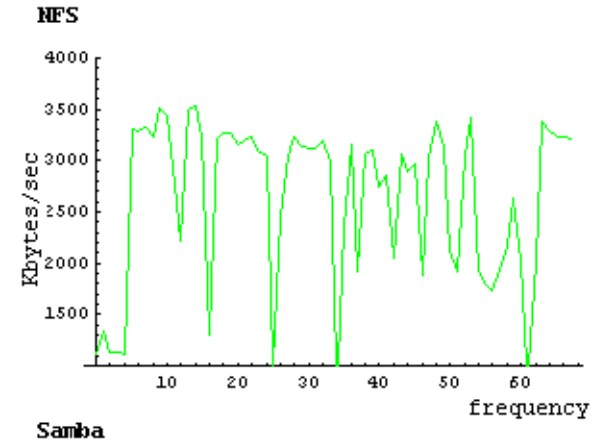

NFS
Kbytes/sec
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency

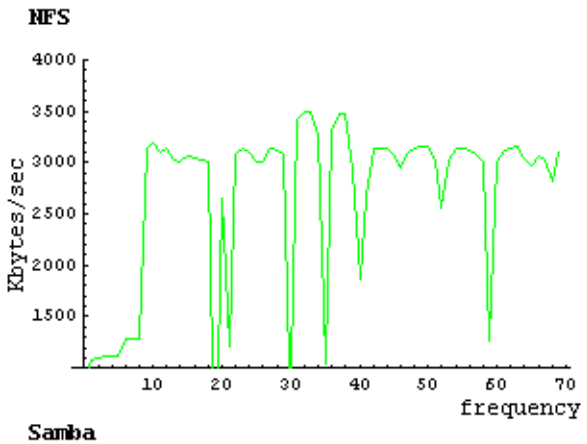

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
2000
10
20
30
40
50
60
70
frequency

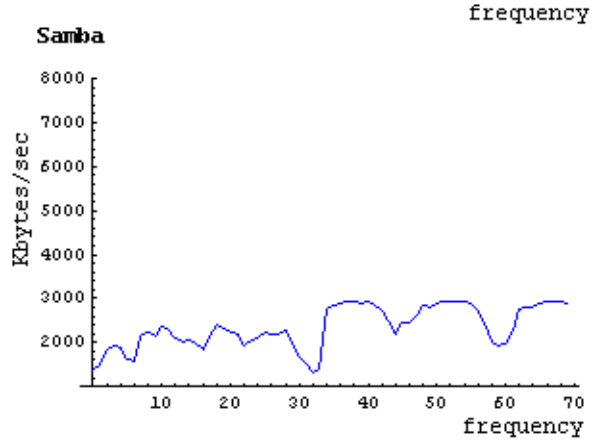

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
2000
10
20
30
40
50
60
70
frequency

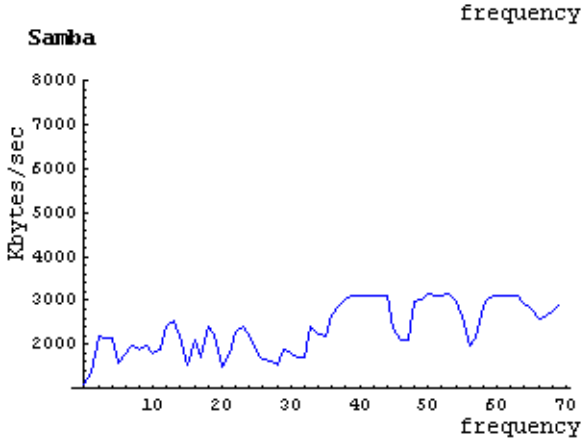

file size = 512 MB
openAFS
Kbytes/sec
1400
1200
1000
800
600
10
20
30
40
50
60
70
frequency


file size = 1 GB
openAFS
Kbytes/sec
1400
1200
1000
800
600
10
20
30
40
50
60
frequency

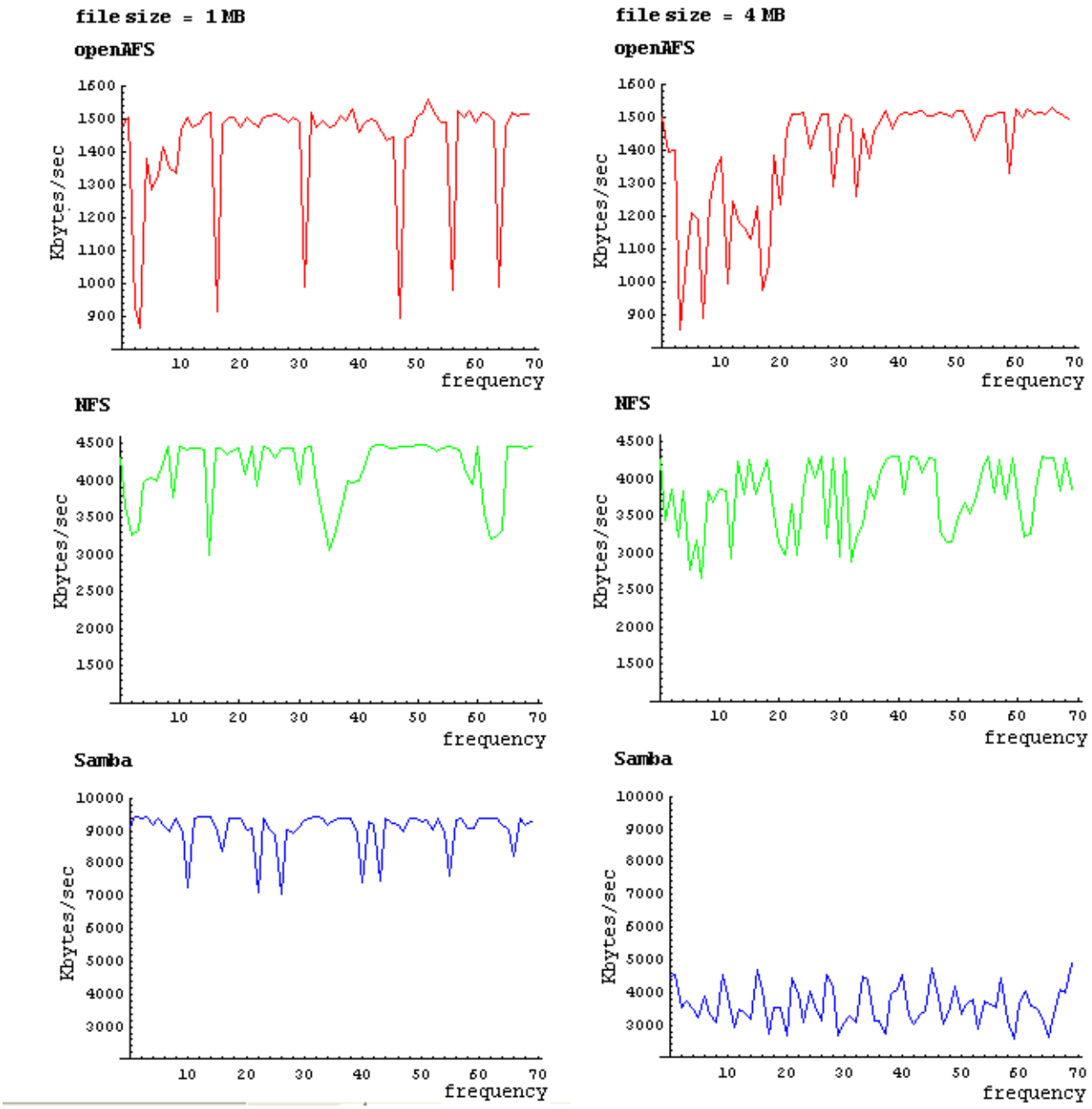

NFS
Kbytes/sec
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
frequency

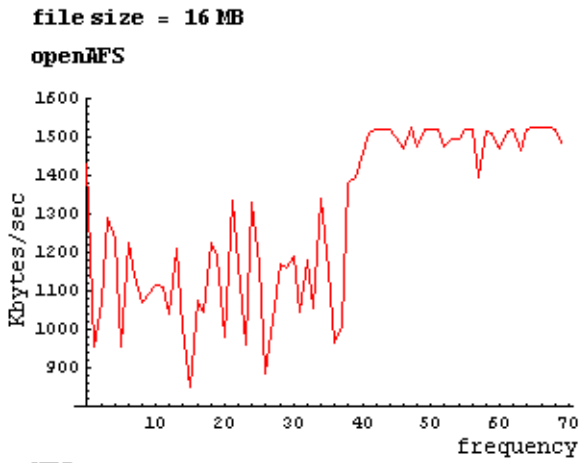

NFS
Kbytes/sec
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency

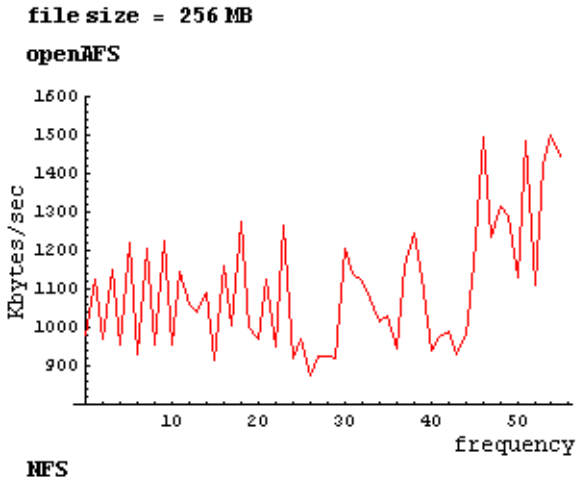

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
2000
10
20
30
40
50
60
70
frequency

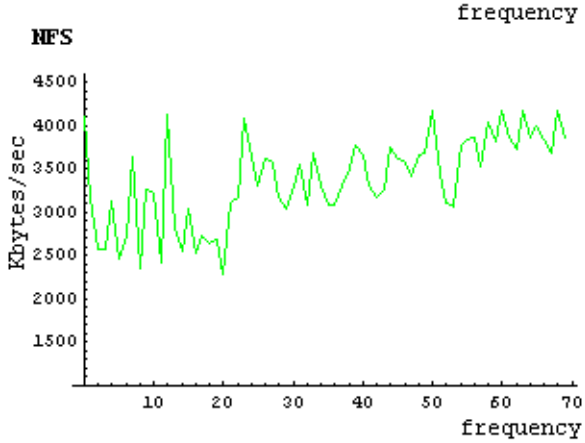

Samba
Kbytes/sec
8000
7000
6000
5000
4000
3000
2000
10
20
30
40
50
60
70
frequency

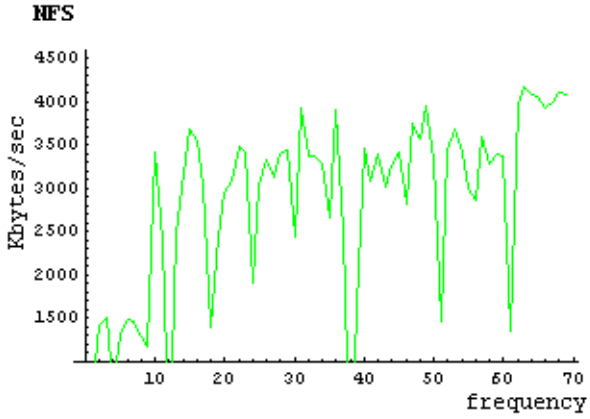
Write performance for client2

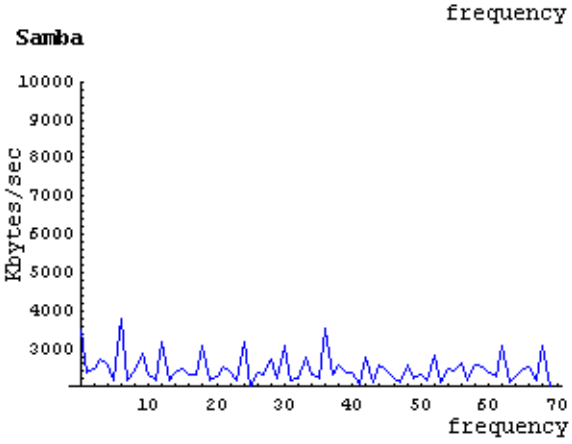


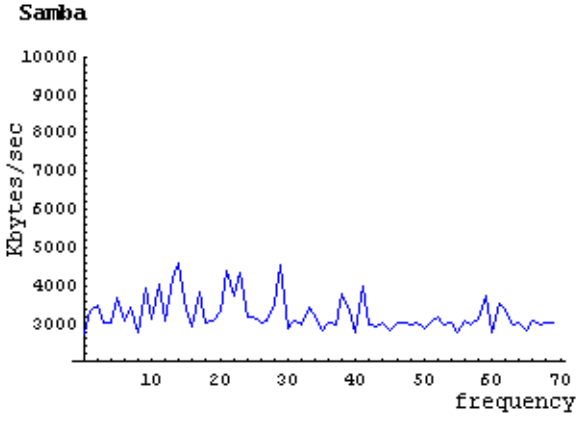

file size = 16 MB
openAFS
Kbytes/sec
1600
1500
1400
1300
1200
1100
1000
900
10
20
30
40
50
60
70
frequency

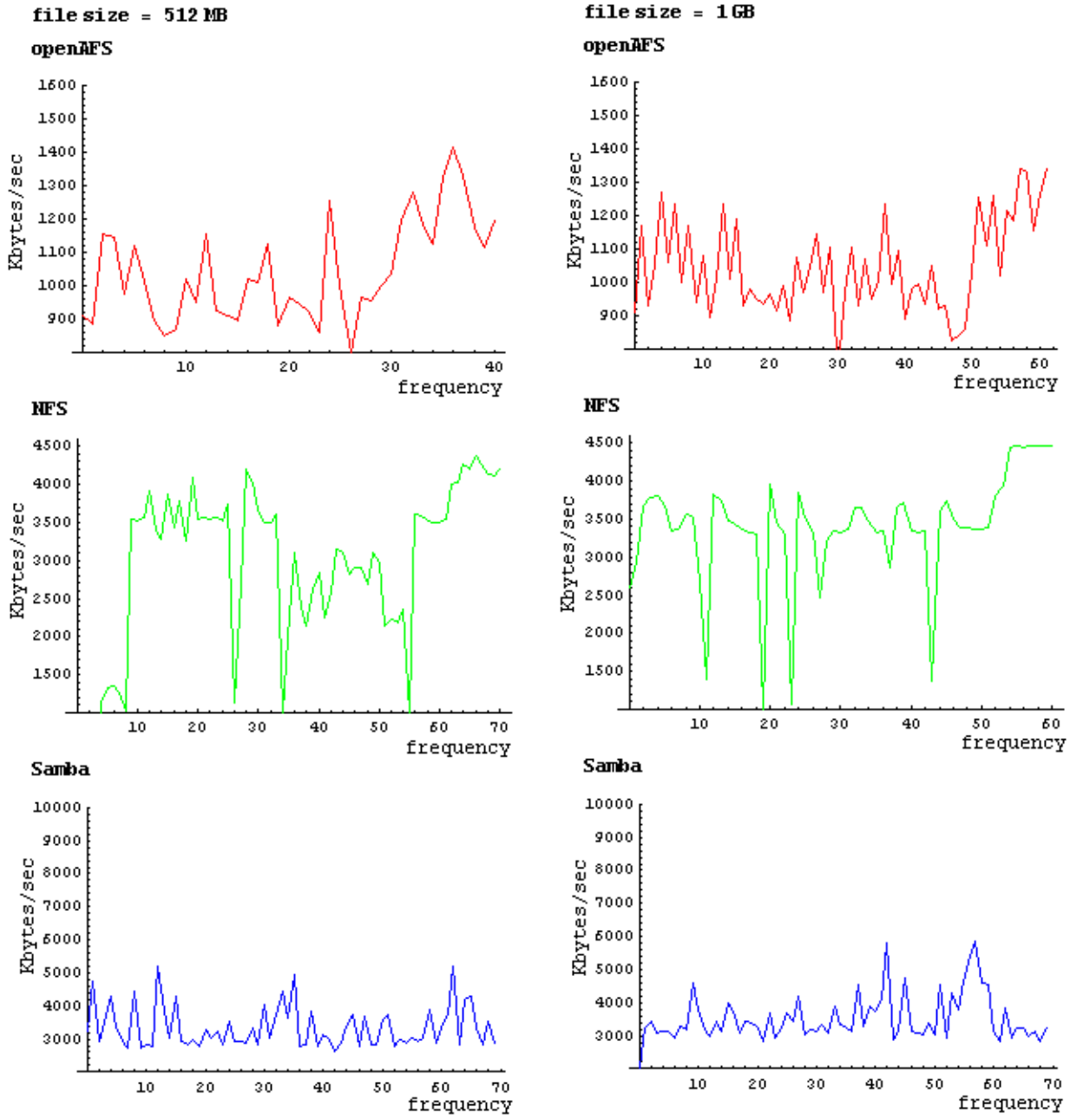

file size = 256 MB
openAFS
Kbytes/sec
1600
1500
1400
1300
1200
1100
1000
900
10
20
30
40
50
frequency


NFS
Kbytes/sec
4500
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency

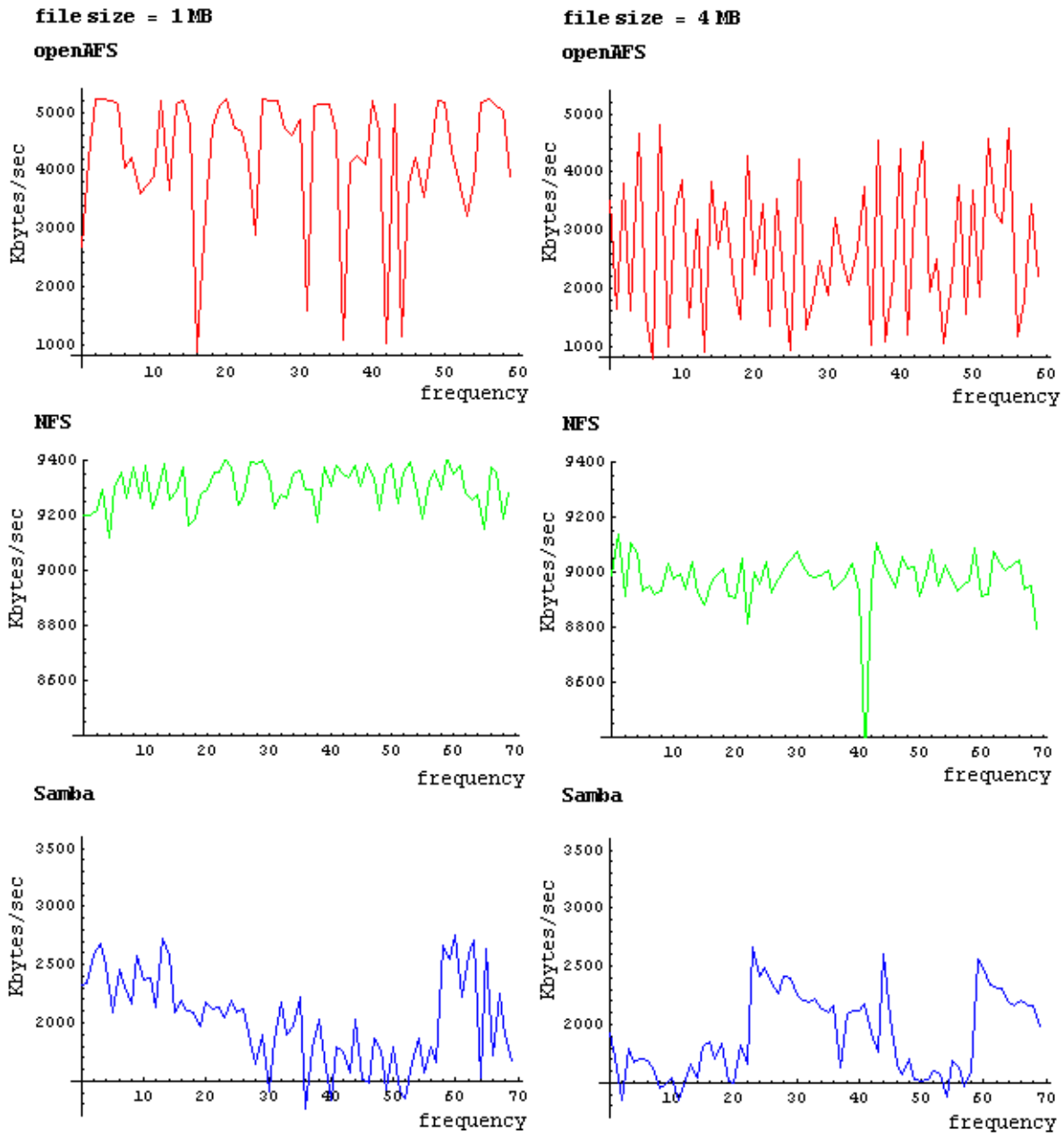

NFS
Kbytes/sec
4500
4000
3500
3000
2500
2000
1500
10
20
30
40
50
60
70
frequency

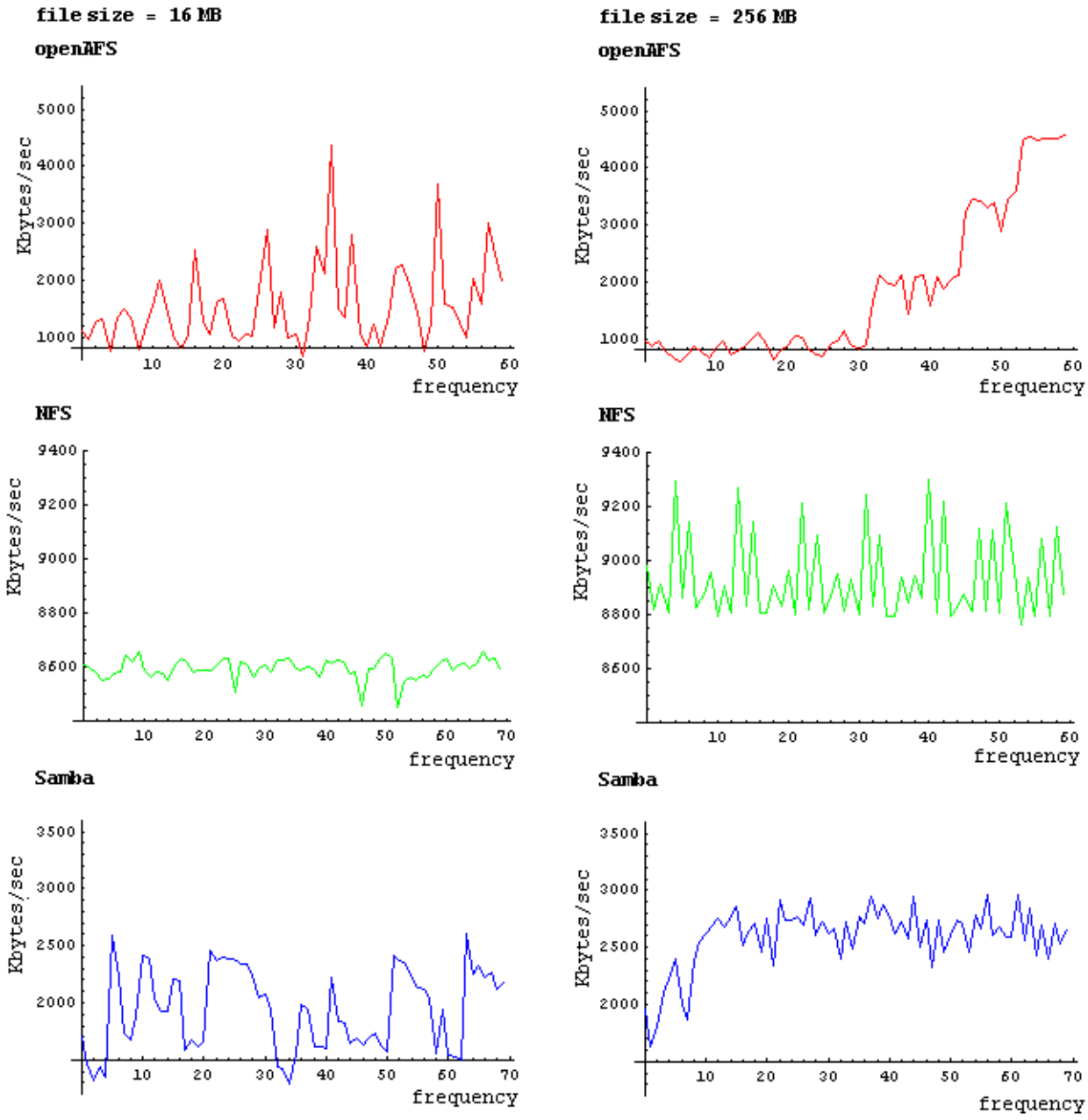

Samba
Kbytes/sec
10000
9000
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

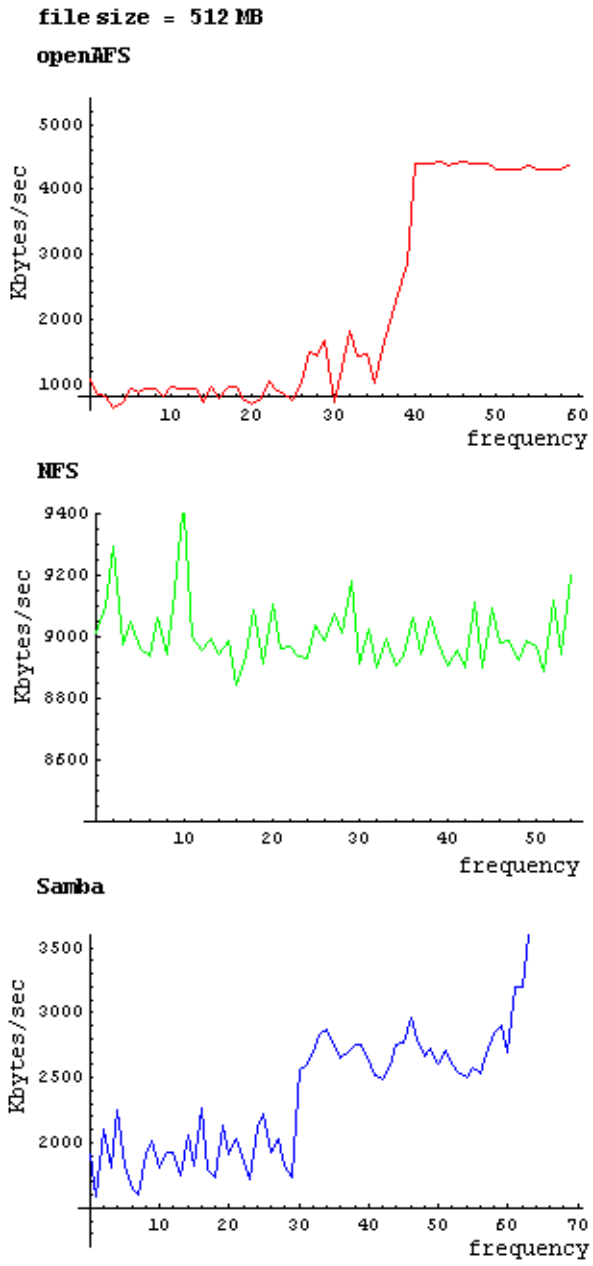

Samba
Kbytes/sec
10000
9000
8000
7000
6000
5000
4000
3000
10
20
30
40
50
60
70
frequency

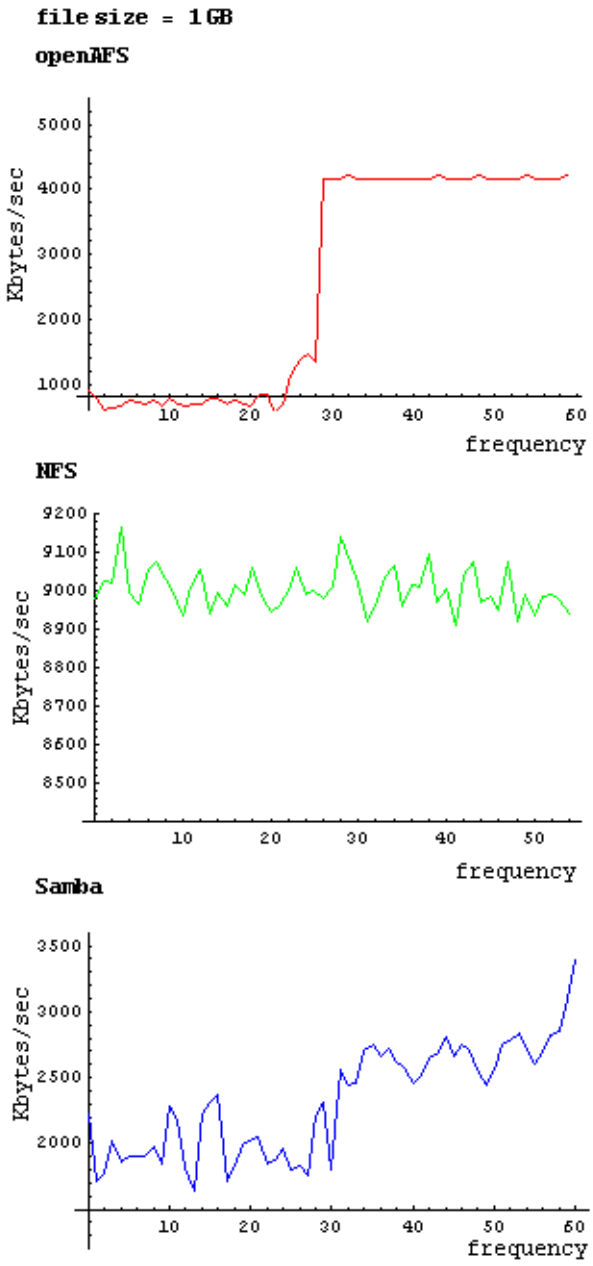

file size = 512 MB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency
file size = 1 GB
openAFS
Kbytes/sec
frequency
NFS
Kbytes/sec
frequency
Samba
Kbytes/sec
frequency

Write performance for *linux client*




file size = 16 MB
openAFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
frequency
file size = 256 MB
openAFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
frequency
NFS
Kbytes/sec
9400
9200
9000
8800
8600
10
20
30
40
50
60
70
frequency
NFS
Kbytes/sec
9400
9200
9000
8800
8600
10
20
30
40
50
60
frequency
Samba
Kbytes/sec
3500
3000
2500
2000
10
20
30
40
50
60
70
frequency
Samba
Kbytes/sec
3500
3000
2500
2000
10
20
30
40
50
60
70
frequency


file size = 512 MB
openAFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
frequency
NFS
Kbytes/sec
9400
9200
9000
8800
8600
10
20
30
40
50
frequency
Samba
Kbytes/sec
3500
3000
2500
2000
10
20
30
40
50
60
70
frequency


file size = 1 GB
openAFS
Kbytes/sec
5000
4000
3000
2000
1000
10
20
30
40
50
60
frequency
NFS
Kbytes/sec
9200
9100
9000
8900
8800
8700
8600
8500
10
20
30
40
50
frequency
Samba
Kbytes/sec
3500
3000
2500
2000
10
20
30
40
50
60
frequency

Appendix E

Write Performance							
client1		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	1616	1569	1260	1244	1385	1240
	stddev	195	231	180	172	177	117
NFS	average	3452	3229	3118	2832	3359	2942
	stddev	176	587	327	788	399	359
SAMBA	average	7006	3630	2641	3049	2687	2836
	stddev	739	762	607	461	309	280

Table 1. Average and standard deviation write performance results on client1

Write Performance							
client2		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	1427	1397	1290	1101	1043	1045
	stddev	171	170	214	164	150	140
NFS	average	4157	3768	3380	2910	2922	3403
	stddev	426	477	507	971	1082	734
SAMBA	average	9058	3617	2479	3250	3340	3516
	stddev	582	588	371	446	639	714

Table 2. Average and standard deviation write performance results on client2

Write Performance							
client3		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	850	785	673	944	1040	1048
	stddev	238	355	210	123	144	176
NFS	average	3470	3223	2982	2563	2643	2704
	stddev	174	232	291	917	786	803
SAMBA	average	6722	3097	2383	2684	2344	2380
	stddev	925	690	662	630	477	573

Table 3. Average and standard deviation write performance results on client3

Write Performance							
Linux client		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	4245	2643	1552	1846	2197	2777
	stddev	1159	1206	737	1311	1593	1676
NFS	average	9305	8975	8594	8933	9007	9004
	stddev	72	102	37	155	106	53
SAMBA	average	2028	1911	1937	2590	2463	2324
	stddev	392	356	357	271	578	417

Table 4. Average and standard deviation write performance results on *linux client*

Read Performance							
client1		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	2117	2029	1588	1479	1626	1421
	stddev	273	388	289	287	264	173
NFS	average	2816	2695	2524	1975	2566	2139
	stddev	121	200	194	540	425	144
SAMBA	average	2682	2423	2206	2207	1830	1810
	stddev	212	322	425	320	268	282

Table 5. Read performance results on client1

Read Performance							
client2		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	1818	1231	921	909	929	827
	stddev	178	238	122	123	269	264
NFS	average	4436	4211	3621	2189	2010	2187
	stddev	356	662	1232	888	721	467
SAMBA	average	8334	8146	8004	5816	4258	3979
	stddev	180	293	331	549	807	942

Table 6. Read performance results on client2

Read Performance							
client3		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	2651	2379	1794	2129	1832	1874
	stddev	115	201	232	272	145	286
NFS	average	3917	3810	3589	2732	3008	2847
	stddev	192	282	290	730	831	676
SAMBA	average	3677	3337	2901	2885	2200	2213
	stddev	281	420	407	398	412	448

Table 8. Read performance results on client3

Read Performance							
Linux client		File Size					
		1MB	4MB	16MB	256MB	512MB	1GB
OpenAFS	average	3014	2279	1586	1654	1956	3476
	stddev	704	738	566	1004	1202	545
NFS	average	9329	8867	8481	8833	8865	8868
	stddev	61	104	35	107	120	39
SAMBA	average	2322	2223	2074	2458	2331	2059
	stddev	549	527	516	341	629	450

Table 7. Read performance results on *linux client*

Appendix F

Raw measurements data

Samba's Raw performance measurements data on *linux client* for 1 Gbyte file

The output that IOzone generates for one measurement is given bellow.

```
Iozone: Performance Test of File I/O
        Version $Revision: 3.239 $
Compiled for 32 bit mode.
Build: linux

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million,
             Jean-Marc Zucconi, Jeff Blomberg,
             Erik Habbinga, Kris Strecker, Walter Wong.

Run began: Wed Apr 13 07:06:49 2005

File size set to 1048576 KB
Record Size 4 KB
Excel chart generation enabled
Auto Mode
Include close in write timing
Command line used: /opt/iozone/bin/iozone -f ./iozone.tmp -s 1G -r 4k -i 0 -i 1 -R
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
                                                              random  random    bkwd
              KB  reclen   write rewrite    read    reread    read   write    read
         1048576       4    2225    1753     1953     1661

iozone test complete.
```

```
Excel output is below:

"Writer report"
        "4"
"1048576"   2225

"Re-writer report"
        "4"
"1048576"   1753

"Reader report"
        "4"
"1048576"   1953

"Re-Reader report"
        "4"
"1048576"   1661
```

Behandlet performance measurements data on *linux client* for 1 Gbyte file

write and read measurements values obtained by running *genFiles.pl* script (appendix B)

write measurements values

```
{0,2225},{1,1709},{2,1772},{3,2021},{4,1865},{5,1897},{6,1898},{7,1901},{8,1975},
{9,1843},{10,2290},{11,2180},{12,1822},{13,1644},{14,2226},{15,2313},{16,2368},
{17,1711},{18,1829},{19,1989},{20,2014},{21,2048},{22,1847},{23,1868},{24,1955},
{25,1801},{26,1827},{27,1761},{28,2212},{29,2305},{30,1806},{31,2566},{32,2443},
{33,2455},{34,2702},{35,2751},{36,2664},{37,2722},{38,2623},{39,2579},{40,2461},
{41,2519},{42,2653},{43,2684},{44,2809},{45,2662},{46,2743},{47,2701},{48,2561},
{49,2447},{50,2555},{51,2745},{52,2777},{53,2834},{54,2716},{55,2608},{56,2693},
{57,2818},{58,2856},{59,3105},{60,3394},
```

read measurements values

```
{1,1953},{2,1741},{3,1549},{4,1923},{5,1573},{6,1737},{7,1444},{8,1614},{9,1861},
{10,2001},{11,1996},{12,1493},{13,1416},{14,1587},{15,1607},{16,1668},{17,1438},
```

{18,1504},{19,1487},{20,1434},{21,1674},{22,1905},{23,1958},{24,1559},{25,1703},
{26,1490},{27,1480},{28,1752},{29,1813},{30,1598},{31,2285},{32,2311},{33,2072},
{34,2560},{35,2435},{36,2375},{37,2306},{38,2616},{39,2511},{40,2340},{41,2343},
{42,2480},{43,2420},{44,2522},{45,2472},{46,2439},{47,2197},{48,2215},{49,2257},
{50,2322},{51,2727},{52,2489},{53,2673},{54,2653},{55,2510},{56,2365},{57,2383},
{58,2366},{59,2529},{60,3445},