The Impact of Current Storage Techniques on File System Design

Ibrahim A.
Dept. of CSE,
IIT Kanpur
ibrahim@cse.iitk.ac.in

Arun Raghavan
Dept. of CSE,
IIT Kanpur
arunsr@cse.iitk.ac.in

Kamal Sharma
Dept. of CSE,
IIT Kanpur
kamals@cse.iitk.ac.in

***Abstract*— Storage techniques have seen a number of incremental changes over the last two decades, starting from single, direct-access disks to intelligent, multiple-disk, object-based storage devices which provide far richer semantics than seen previously. This paper tries to track and understand the corresponding changes seen in file system design, and provide some insights into what the future might hold in this area.**

I. Introduction

The storage landscape has seen a slow but definite change over the last two decades. Slow and bulky disks gave way to smaller, faster disks, which have now started to reach to the technical limits of mechanical storage. In the mean time several techniques have been formulated to improve the speed and reliability of these devices (buffering and RAID [20], for example). In the mean time, requirements have driven the basic architecture of storage systems from centrally-located single-user storage, to centrally located multi-user storage, to tightly-, and even loosely-coupled, distributed storage.

In this paper we analyse the various file system technologies that have evolved to address the special requirements of each of these storage technologies. Some fundamental requirements of a storage system that we look at in each scenario are listed below:

1) *Caching/Buffering*: Conventional direct-attached storage is orders of magnitude slower than higher memory hierarchies, and the storage technology is reaching the limits of mechanical technology. Fast, dedicated solid-state memory, as well as primary memory, aer used as caches to hide the speed of the underlying technology. If the system relies on general-purpose networks, which usually have high latency, delays are even more pronounced, and caching becomes even more important.
2) *Locking*: In order to protect concurrent access to files, locking mechanisms are often provided by file systems. The problem, while not too complex on centralized systems, becomes far for difficult in a distributed environment.
3) *Access Control*: In a multi-user environment, it is important to provide mechanisms to choose which users who can access different data. In traditional Unix systems, for example, access-control is provided in the form of file/directory permissions.
4) *Security:* Some storage systems now traverse untrusted networks, or contain highly sensitive data. In such cases, the file system might need to provide mechanisms such as digests and encryption.
5) *Disk Allocation Techniques:* These have not been analysed in our survey to a large extent as we were not able to find suitable literature. However, object-based storage devices do provide some change in this area.

II. Direct-Attached Storage

The traditional method of having disks that are directly connected to the server machine (direct-attached storage, or DAS) has not seen major breakthroughs in almost two decades. Most of the improvements have been focused towards reliability, using such technologies as replication and journaling (a concept borrowed from database systems). We examine another idea for file-systems that has been borrowed from database systems.

A. Log-Structured File Systems

The challenge in building high performance file systems lies in using the disk system efficiently. Nowadays large caches go a long way in absorbing read traffic, and buffers do the same for write delays. Achieving efficient writes to disk implies writing data in large, contiguous units. The central idea in log-structured file systems is that by aggressively caching data and applying database logging techniques, all disk writes can be made sequential.

When we consider the operations on small files, most of the disk I/O seen is for accessing metadata. In many common file systems, six distinct I/O operations may be required to create a new file (create an inode for the new file, read the directory inode, write the data of the new file, update the inode and block bitmaps). Log structured file systems (LFS) solve this problem to an extent by reducing the number of I/O operations corresponding to the metadata for the creation and writes for small files. The LFS has only one data structure in the disk apart from the super block – the log. So the LFS writes all the file system data sequentially in a log-like structure. A log consists of a series of segments where each segment contains both data and inode blocks. Traditional file systems like Ext2 [6] usually write inode blocks at a fixed place on the disk, causing overhead due to disk seeks. A log structured file

system gathers a segment worth of data in memory (which will be hundreds of KB or MB) and appends the segment at the end of the log. This dramatically improves the write performance in the case of small files, while maintaining the same read performance.

The main challenge while using an LFS is to have large amounts of contiguous data available at all times. This is usually handled using a segment cleaner to regenerate large free extents. If there is available free space, the cleaner coalesces that free regions to produce clean segments. The cleaner can run when the disk is idle, so that regular file accesses are not affected; however during periods of high activity the cleaner may also be run along with normal file accesses. Depending on the file access patterns, cleaning can potentially be very expensive and can degrade system performance. Thus for LFS the key issue is the cost of cleaning. This, and the general issue of designing and implementing an LFS may be found in the seminal work on this topic by M. Rosenblum and J. Ousterhout [23].

III. Traditional Network-Attached Storage

As the traditional computing model of dumb terminals connecting to a centralised multi-user server gave way to more powerful and independent client-side computing, the need was seen for having a central store that was visible to all these clients. Network-attached storage (NAS) technologies were developed and evolved for this purpose. The most common of these are Sun's Network File System [24] and the Common Internet File System [9], which are popularly used in small- to medium-scale deployments. We first present the general challenges in NAS file system design, and mention, alongside, the how NFS and CIFS address these.

A. Locking

Locking can generally be classified in three ways [25]. Exclusive locks allow only a single process to hold a lock, while shared locks allow multiple processes to simultaneously lock a file. Record-locking refers to the ability to lock certain sections of a file. Mandatory locks are enforced by the file system, while with advisory locks, the choice of cooperation and honouring of the lock is left to the application

Typically, a user-level daemon is implemented for the NAS file locking mechanism, since these systems are generally stateless. A client will request a lock from a server – a daemon running on a server. It might also maintain a monitor program to keep a tab on the state of the server. When both of these processes give a favorable reply, the client assumes that the file is locked. The server also has a monitor program to check the status of the client. This is typically done to handle various scenarios where a server or a client crashes [29].

NFS, implements a Network Lock Manager (NLM) for locking because it is a stateless protocol. NFS, till version 3, support only advisory locking of files and byte ranges. A status monitor, called `statd`, runs on the server as well as the client for handling crashes of the other. When the NFS implementation on the client receives a lock request, instead of the usual NFS RPC, an NLM RPC call is made.

CIFS defines three types of locks – Exclusive locks, Batch Locks and Level 2 locks. Exclusive locks have already been described. Batch locks help reduce the network traffic when multiple open/close operations are done on a file by a single client – the server keeps the file open, even when the local access by the client is finished, until no other client requests for the file access. Level 2 locks are introduced for reading. Multiple readers may hold the lock simultaneously, but when a write lock is requested, the server informs all clients, which then discard read-ahead data, and reread the data from the server.

B. Buffering and Caching

Typically, each process needing a resource from the server requires a fetching of attributes/data from the server. If the data is read/written using block level access on the server, the caching methodology can be similar to the traditional direct-attached storage system. On the client side there are asynchronous threads running which perform the fetching of the data from the client. These threads often prefetch the data in a manner analogous to the technique applied in direct-attached storage.

Caching can be used to for various file system objects [25]:

1) *File Attribute Caching*: In many situations, only the attributes in a file are accessed like owner, permission, etc. The attribute data size is typically small and as it is accessed frequently, caching metadata greatly improves performance of the system. As the attributes are stored on the client itself, multiple requests are absorbed by the cache and not sent to the server, thus reducing the network overhead. When the client changes any data it may immediately be written to the server (write-through cache), or may be retained for a while before being committed to disk.
2) *Client Data Caching*: Clients often do read-ahead and write-behind to improve overall performance. When the client receives data it stores it in a buffer cache until it needs to be evicted or is invalidated by a write to the same region.
3) Server-Side Caching: In this case caching is done by the server, making client request processing faster. Also, multiple requests may access the same data from the server. In that case, the server can benefit from caching this data
    a) Inode Cache: Contains the file attributes recently accessed.
    b) Directory Name Lookup Cache (DNLC): This NFS-specific cache holds the information about recently fetched directories. This makes access to the file system's frequently accessed data faster.
    c) Buffer Cache: This cache is similar to the file buffer cache is used in local system for maintaining file data.

In NFS, for a read operation a RPC call is made on behalf of the client. The client threads make read prefetch RPC calls. For write operation, the write is done on the buffer cache and when this is full, async threads write the data to the server through the more RPC calls.

In case of NFSv2 [18], the write RPC calls would not return till the data is written to a stable storage device. However, in NFSv3 [5], this condition was relaxed and it was the client's responsibility to give a commit signal after the entire file operation was completed. File attributes in the NFS typically remain in cache for around 60 seconds and they are flushed out.

Caching in CIFS is mainly dependent on the locks acquired by the client. In the exclusive lock mode the client can cache the entire file and flush out the data when it has completed all the operation. In other CIFS locks, caching is done for the read operation. The server controls the state of the cache copy of all clients – this differs from the stateless paradigm used by NFS and several other NAS protocols.

C. Security and Access Control

We talk about the following aspects of security and access control

1) *Authentication*: Verifying the user's credentials before permitting access to the system.
2) *Authorization/Access Control*: Controlling what data can be accessed by a given user or group of users.

Encryption, which we do not address, can be handled at two levels, depending on the requirements – encryption of the transport, and encryption of the stored data itself.

Typically, the NAS server will maintain an Access Control List (ACL) which verifies the user authorization.

NFS employs an RPC security mechanism (`AUTH_SYS` authentication protocol), where each RPC requests contains the UID and GID to which the client belongs. It may also use Kerberos-based user authentication [19], which is often the preferred method.

The NFS protocol works by giving file handles to the authenticated client. It can be shown that on some untrusted networks where attackers are present, file handles can be broken and attackers can directly communicate with the NFS daemon. In [24], Traeger et al. present a method of using random file handles to prevent this problem.

CIFS provides share level and user level access control. A common access password may be set for the file in the former, where the later uses UID for authentication.

D. Direct Access File System

The technologies seen thus far are relatively mature and stable. However, with the ongoing trend of huge data requirement for the business environment, these file systems have limited performance. The Direct Access File System (DAFS), introduced in [15], proposes a new protocol using Remote Direct Memory Access (RDMA) and an architecture to provide high access low latency over Gigabit Ethernet and InfiniBand networks.

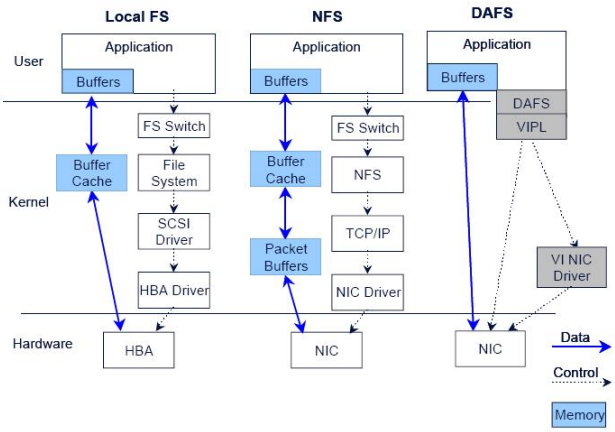

Fig. 1. Overhead of DAFS when compared with traditional approaches

The driving principle behind the design of DAFS was the fact that current network file systems operate on such interconnects as IP-based networks. This introduces performance bottlenecks due to three reasons. Firstly, every transmission traverses several protocol layers, each of which introduces it's own processing and data overhead. Secondly, as the data traverses multiple layers, multiple memory copy operations may be required. Finally, on a slightly different note, DAFS is a user-space file system. This means that it is, to some degree, independent of the operating system, and can be modified without needing to modify the operating system kernel. A comparison of the overhead of DAFS when compared with more traditional approaches is shown in figure (1).

In order to avoid the problems caused by traversing multiple network layers when going over the network, DAFS uses RDMA [2]. RDMA is a feature implemented on the network interface card (NIC), or host-bus adaptor (HBA), which allows safe user-space network access, thus bypassing several kernel layers during network communication. A study on such copy-avoidance technologies is presented in [4].

An interesting variant of DAFS is Optimistic DAFS, proposed in [16], which allows the client to access remote memory pages of the file and VM cache exported by the server. This kind of setup is useful in a distributed client environment, and is illustrated in figure (2).

The cache directory may be distributed among servers, and clients can get handles that refer directly to the server cache. Clients are not notified at the time of cache invalidations. Whenever there is a memory access, the remote cache is checked – if the reference is valid, the corresponding data is sent, and if it is stale an exception is marked and the data is transmitted through the normal RPC method.

IV. Wider-Area Storage

In the last decade, the demand for scalable shared storage has risen dramatically, with the increased use of high-availability servers, high-performance computing, computing clusters, and shared storage across installations with upto tens

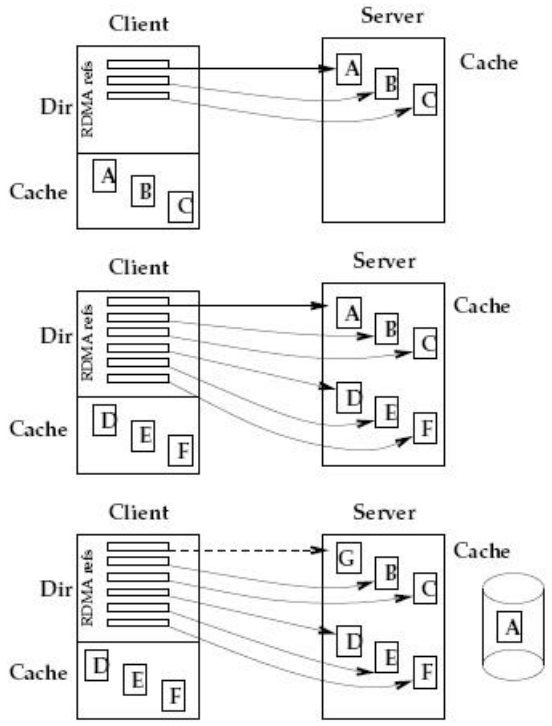

Fig. 2. Architecture of Optimistic DAFS

of thousands of users. In the mean time, general-purpose network technology is approaching the speeds of conventional storage interconnects. These two trends together have heralded an era of highly scalable storage systems which are able to use commodity networking components (this trend is further examined in section IV-F.

We first briefly analyse two storage techniques that are popular in several scenarios – Distributed and Cluster File Systems. We then see a newer technology to address the problem of sharing data across a WAN type environment. We conclude the section by discussing two promising new trends/technologies in this area.

A. Cluster File Systems

Cluster file systems provide a distributed, scalable architecture by replacing the traditional file server by a group of connected nodes using some interconnect. The group of server nodes forms a tightly coupled cluster where each node can share and access the common storage. The elimination of the single server bottleneck improves performance as the clusters share the total workload. It also adds to the fault tolerance – if one of the server nodes fails, another one can cater to its requests. Data consistency is maintained with the help of a common metadata server which keeps the information about the files. This can be replicated for improved availability. Each node has direct access to the shared data over the network. The cluster filesystem acts as a software layer that enables the cluster nodes to share data and work as a single unit.

A typical example of a cluster filesystem is the Lustre File System [7]. It uses object-based disks (OBDs – these can be virtual, or actual object-based storage of the type covered in section IV-E) for storage and metadata servers (MDS) for storing filesystem metadata. The actual file system I/O and interface with the storage devices is done by a set of distributed Object storage targets (OST). Whenever a client want to create a file, it contacts the metadata server, which creates the inode and contacts the Object Storage Target to create the objects that will hold the file. So objects that hold the data and they can be striped over multiple OSTs. The data is actually read and written by the OSTs on to the actual storage. The metadata server is updated only when there are any namespace changes associated with the new file.

The object storage target links the client request to the underlying physical storage, represented by the object-based disk. The OBDs need not limited to actual disks, as the interaction of the OST with the actual storage device happens through the device driver. The functionality of the device driver hides the specific identity of the underlying storage that is being used. Thus, Lustre can use any of the existing Linux file systems by providing a OBD driver of that filesystem. Apart from the storage abstraction this provides, addition of new OBDs to the OSTs is also made possible without affecting the nodes.

Lustre provides security by integrating itself with any of the existing security systems. Depending upon the level of authentication, authorization and security required, the Lustre can use specific security systems, without making changes in the Lustre itself. It uses the Generic Security Service Application Programming Interface (GSSAPI), an open standard that provides secure session communications supporting authentication, data integrity and data confidentiality.

B. Distributed File Systems

Distributed File Systems (DFS) are more loosely coupled than cluster file systems – each server talks has it's own storage, and the DFS presents an aggregated view of all this storage, while handling matters such as replication across servers behind this abstraction. In this manner, having a number of servers instead of a single file server greatly improves the reliability of the entire system. There may also be more effective space utilization among the clients, since is often better to have large space given to the server than all the clients.

The DFS implementation is typically part of the operating system. The DFS is a transparent layer and clients are able to access the files just like they would have done on a local file system [11].

Caching, as with other file systems, is essential to improving the performance of a distributed file system. The distributed nature of the system, however, requires that we tackle the problem of cache coherence. There are two basic approaches to this:

1) *Client-initiated invalidation*: Here, the client checks for cache consistency with the server at regular intervals.

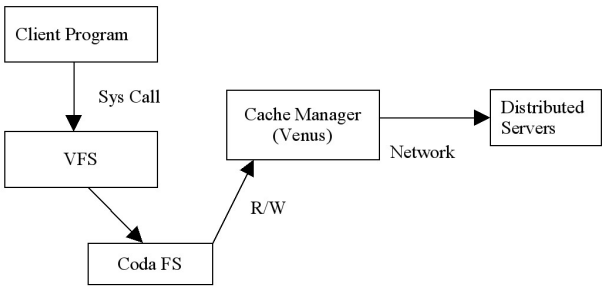

Fig. 3. The Coda implementation

Checking at shorter intervals allows us to make better coherence guarantees, but degrades overall performance due to network bandwidth consumption.

2) *Server-initiated invalidation*: In this case, the server maintains the information about the cache state data held by clients. On update to the data, the server notifies all the potential clients which have the data, which then need to invalidate the corresponding entry (the entry could also be updated, but this does not provide sufficient performance gain to justify the bandwidth usage).

We now look at the design of some distributed file systems. The Andrew File System (AFS) was implemented for distributed academic environment, with scalability and security as important concern. For security, the AFS uses Kerberos mechanism. AFS implements a server-initiated approach for cache consistency. When a client needs to discover the actual location of some data, it does so by using the "volume location server" in the appropriate cell (a group of AFS servers in a single administrative domain).

The Coda file system [3] is a descendant of AFS. The client connects to a group server rather than an individual server. The basic working of Coda is illustrated in figure (3). Coda has a cache manager, named Venus. The figure shows what happens during the system call when opening a file. The client makes the system call, which is handled by the kernel's Virtual File System (VFS) layer. The VFS translates the request into the appropriate file system call in this case the Coda "open" call. The cache manager then checks it's cache for file, and if it is not present makes an RPC request to the Coda servers.

One of the most distinctive feature which differentiates Coda from AFS is the availability of files to clients without being connected to the DFS. In the case where clients are disconnected from the DFS network, Venus does hides the fact that the actual Coda FS is not available – instead it maintains a log of all the modifications to the file. Later, when the client is back into the network, the Venus manager will replicate the necessary changes on the server in order to maintain the consistency. Coda allows some application-specific methods to handle conflicts while committing (they call it "reintegrating") these modifications – failing this, a method for human intervention is provided.

C. Wide-Area File Systems

The techniques described so far do not scale well to WAN-range networks. Wide-Area File Systems (WAFS) are used to address this problem. The WAFS solution combines the distributed file system approach along with aggressive caching technologies. The WAFS can work with the current NFS/CIFS server in the organization, minimising the number of infrastructure needed during migration. Typically, there exists a file system redirector between the client and the central sever while acts as a transparent layer, as is the case in WireFS [12]. Caching of data at remote locations for read/write enables to have huge performance impact. Co-operative caching, as described in the WireFS work, reduces the caching bottleneck at the server.

Typically, a file-aware protocol needs to be used between the client and the server so that only particular changes are sent instead of the whole file. The read performance is increased by the cache, to hide latency. For write buffering, either write-through or write back approach can be adopted. With the help of user access privileges, coherence can be guaranteed. For writing to a file, the write permission has to be sought from the central sever. This prevents any conflicts between the clients. Immediately after the update is made, it is reflected at all sites, so that there isn't any dirty data on the network.

Commercially available WAFS-based solutions include TacitNetworks' Tacit Datacenter Ishared Serve (server) with Remote Ishared Appliance (client), Cisco's Actona Subsidiary, and Riverhead Technology;s Steelhead appliances.

D. Serverless File Systems

The approaches described so far rely on a central server, or servers, to cater for the needs of the clients. In contrast to the this the serverless file systems [1] share the workload among the various workstations cooperating as peers to provide all file system services. Each of the peers can have control any of the data blocks, store it and cache it. If a component fails, the serverless architecture helps in distributing the responsibilities of the failed node to it's peers. The performance measures for an implementation of serverless file system called xFS [28], with 32 nodes, showed that each client receives almost as much read and write throughput as it would see if it were the only active client.

The basic architecture of the serverless file system is the sharing of storage, cache and control over the cooperating workstations. There were three main factors for the development of work in this area: the opportunity provided by the fast switched, the expanding demands of users and the disadvantages of a central server system. In the case of serverless file system, there is no central bottleneck. xFS, for example, dynamically distributes control processing across the system on a per-file granularity basis, by utilizing a new serverless management scheme. It distributes its data by implementing a software RAID, using log-based network striping similar to the Zebra file system [10]. Caching at a central server caching is replaced by cooperative caching, where portions of client memory form a large global file cache.

The file management scheme is such that the all the file access details (which change dynamically) are stored in each of the peers, so that any peer can access any file using the management scheme. The managers also check the cache consistency state of each of the peers. For the disk layout xFS uses a combination of the log structured file system mechanism introduced in section (II-A), along with the RAID for striping. The LFS mechanism also helps in crash recovery.

E. Object-Based Storage

A recent introduction in the storage landscape is the concept of object-based storage devices [17]. The basic idea is to move the parts of the file system that deal with the actual storage management, such as block allocation and free block management, on to the storage device itself. The primary benefit of this is that if adequate security mechanisms are provided by the device itself, in conjunction with the typical NAS-type server, clients can directly talk to the storage device, reducing some levels of indirection and thus improving performance. An additional benefit of having a device that provides richer, more powerful semantics than the primitive block-based storage is that the file system designer now needs the only file management aspect of file system design, rather than messy disk-related details.

1) OBFS: In [27], an implementation of an Object-Based File System (OBFS) is presented. Here, the object-based storage device (OSD) is used almost as a drop-in replacement for the typical block-based devices – blocks are mapped directly to objects. The reason for this is that striping files across devices is far more efficient, providing better parallel access. The salient features of their implementation are given below. They have contrasted their implementation to Ext2/Ext3, and XFS [26]. We do not include their performance metrics as these are done with the `-o sync` option of the `mount` command, bypassing the benefits of the Linux kernel's aggressive caching. They do this because their first implementation, used in this paper, is based entirely in user space. There has been as subsequent in-kernel implementation described in [13], but this does not include a strong caching framework.

1) Both implementations described above were accomplished with about 2000 lines of code. This does not include the caching framework, but does serve to highlight the simplicity introduced by having a storage device with more powerful interfaces.
2) Since the OSD provides a flat namespace for objects, OBFS uses hash-tables for representing directories, instead of B/B+ trees, which are used in most production file systems.
3) Since OBFS is expected to be striped across OSDs, the blocks stored on each device are likely to not have predictable locality.
4) An important feature of OBFS is the dichotomy of small and large blocks. OBFS divides the storage into "regions", some supporting large blocks (512KB), and some small blocks (4KB). This is done so as to provide efficient storage for both large and small files, without losing too much disk space to fragmentation.

We find two significant inferences from the OBFS work. Firstly, the OSD interface serves to significantly reduce complexity of file system design (or at least the file management part of it). Secondly, we believe that this direct mapping of blocks to objects is less effective than mapping the actual objects used by the file system itself (files, directories, and attributes). The only benefit OBFS expects to derive is from parallelism, and we surmise that a cleaner solution is to let the OSD manage parallelism by sitting at a higher level (an analogy would be a RAID controller as opposed to a disk controller). This, we believe, would be truer to the object-based storage paradigm.

2) Lustre and Object-Based Storage: The Lustre cluster file system is particularly well suited to use with object-based storage, since it uses exactly the intended architecture that OSDs are targeted for – clients first look up metadata on a metadata server and are then provided a handle (device, object ID, security permissions etc.) to access the data directly from the device.

F. Virtualisation Frameworks

The current trend in storage technologies is a move away from vendor-specific proprietary systems towards systems consisting of standards-compliant, commodity off-the-shelf (COTS) components, termed "storage bricks". In this section we examine the Locus framework presented in [14], which tries to address the file system design challenges that accompany this trend.

Locus is a low-level framework intended to simplify the creation of cluster-based storage systems based on the "brick-based" described above. It follows the popular "shared virtual disk" model [21], where the upper layers of the file system view the common storage as a single, large store, and the lower layers abstract away the details of the underlying storage, interconnect, fault-tolerance, caching etc. One of Locus' key strengths is that it is modular and flexible, allowing part of the implementation to lie on the application nodes (the file server, typically) and part of it on the storage device itself. The flexibility offered by this method allows the system to be modeled independently of the technology, but adapted to newer techniques as they are available. This is illustrated in figure (4) – each module provides it's own set of expressive semantics, and a directed-graph of modules is created to implement the required system.

For example, consider block allocation, which has traditionally been a part of the file system itself. In the Locus framework, this is yet another module, which can reside on the server node, as it would in a traditional file system, or on an object-based storage device, which has the block-allocation intelligence built into it. Note how this flexibility extends itself neatly to hardware or software RAID, encryption, and volume-management, to name but a few oft-used features. Another important simplification introduced by Locus is the in-band handling of locking. Unlike other systems which have out of

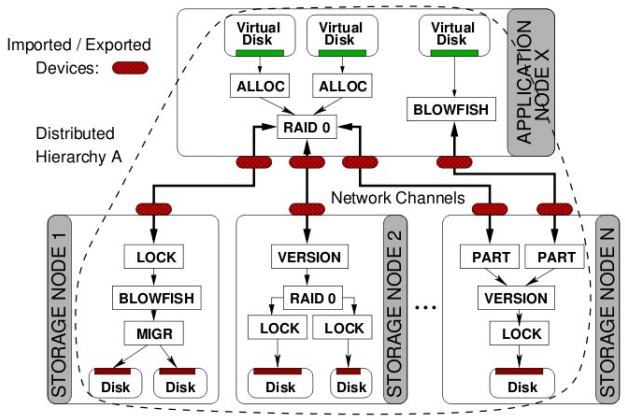

Fig. 4. Example architecture – the Locus framework

band lock management (for example, NLM for NFS which we have seen), locking is implemented in the block layer as yet another module by Locus.

We briefly describe Locus' locking mechanisms in order to provide a flavour of the power of having stackable, semantically-rich modules.

1) Locking in Locus: Locus provides byte- and file-level locking mechanisms. The byte-level sharing module maintains metadata which is basically list of unlocked regions. In order to ensure mutual exclusion to locked regions, a single instance of the locking module services locking of a particular byte ranges. Several instances of the module, each with a separate byte range to service, can be used to balance the load. A useful optimisation, particularly on object-based storage devices, is to map these locking modules onto locking mechanisms provided by the OSD.

In order to leverage the power of the framework, the authors of the Locus paper wrote a user-space file system – Locus-fs – to provide locking primitives. This merely translates the file-level calls to block-level calls, leveraging the locking mechanisms provided by block-level layers. In fact, the block allocator layers also use the same mechanisms to ensure metadata consistency.

The modular design of the Locus system is bound to introduce some overhead of communication between layers. The Locus paper observes a significant difference (~20 microseconds vs. ~300 microseconds) in overhead due to calls over the network on a Gigabit Ethernet backbone. This might be alleviated by faster interconnects, but is still a bottleneck. The system shows good scalability as well, tracing the number of nodes closely. However, these experiments were performed on a relatively small system (maximum of 8 server nodes and 8 storage nodes). The interesting point to understand here is there is a trade-off between the simplicity and scalability of such a system and the overhead introduced. We were not able to find any existing work on this subject, possibly because the methodology is relatively new.

V. Workload-Based Analysis

Through this paper, we have tracked the development of storage systems and the corresponding changes in file system design. We now take a brief look at the possible impacts of some workloads on the both of these.

One of the challenges of analysing workloads and their impact on storage and file systems is that these systems, while designed in a generic framework, are deployed in various, significantly different scenarios, each with disparate requirements. For example, a web server often accesses a set of small files repeatedly and infrequently performs writes, while a database server might have diverse access patterns, depending on the types of queries, and caching strategy of the server software.

We summarise the findings of the study by Roselli, et al [22]. Their work primarily focuses on a university environment with several users accessing a NAS-based storage server. They inserted kernel hooks to get traces of file accesses from Unix- and Windows NT-based systems over some months. The data collected is representative of several typical scenarios of deployment. The findings are largely independent of the underlying technology.

1) *stat()*: The percentage of `stat()` calls, of all file system calls in these workloads was around 97%. In addition to this, `stat()` is often invoked on several files in the same directory in a short span of time. It would thus benefit performance if the file system kept files' metadata structures close to the directory and each other. The NFS `READDIR2` call provides exactly is optimisation.
2) *Block lifetimes*: The patterns in block lifetimes are interesting. The traditional rule of thumb for block lifetimes is that blocks tend to be deleted within the first 30 seconds of their creation, or a long time after that. The Roselli study shows that under some loads this is closet to 5 minutes, while in others there is no such pattern at all. The key observation, however, is that a large number blocks are deleted due to overwriting of the corresponding file – 51% in the lowest workload, but 86-97% in the others. Moreover, a small number of files, less than 5% of all files involved, caused *all* block overwrites [1]. The implications of these findings are listed below.
    a) The rule-of-thumb block lifetime of 30 seconds might not be valid in such scenarios.
    b) Caches that can buffer writes for a long time would be useful in performance enhancement. For reliability, the use of NVRAM, reliable memory, or logs is suggested [8].
3) *Write delays*: It is necessary to examine the effects of the write delay introduced by buffering. The first concern is calls such as `sync()` and `fsync()`. Using methods

[1]Our conjecture is that this is due to the fact that typical POSIX-style file semantics do not allow insertion of data in the middle of a file. Most applications work around this by overwriting the file with an updated copy.

such as NVRAM, we can maintain the system-call semantics without actually flushing to disk. Secondly, the Roselli group has shown that a 4-16MB write buffer provides a sufficient approximation of an infinite buffer. This may not be applicable in several scenarios today (6 years after their study was published), but the key observation is that the required buffer is reasonably bounded, and the argument is aided by falling memory costs.

4) *mmap()*: A major trend in modern application is the replacement of traditional file `read()` and `write()` calls with the `mmap()` method, which maps the entire file into the process' address space. In fact, about 68-85% of all files that were accessed were accessed by this method. Unfortunately, the Roselli study was not able to capture access-patterns when files were accessed in this way. This information might provide some interesting insights.
5) *File sizes*: Traditionally, it has been seen that most file accesses involve small files, and few big files. This trend is generally still prevalent (the study saw 63-88% of access to files less than 16KB in size), but in accordance with today's applications, the number of large files accessed is increasing. File systems today strike a balance by being able to access blocks of small files using direct pointers in their on-disk structures (usually inodes), but using extent-based allocation to allow efficient access to large files as well.

VI. Future Work and Conclusions

The primary observation of this survey is that network-based storage has been, and currently is, the foremost driver of changes in file system technology. Another interesting trend to note is the movement, at least in research circles, from kernel-based file system implementations towards user-space implementations. The simplicity, reliability, and, most importantly, ease-of-use are probably the driving forces here.

Object-based storage techniques represent a paradigm shift in storage methodology – it will be interesting to see the direction these devices take in the future. Additionally, although current file systems often try to use caching enhancements to improve the performance as the ultimate bottleneck are the disk speeds, but the efficacy of this depends on usage patterns.

Currently, there is no implementation of self-adaptive filesystem for workloads that we know of. Future research work in this direction may be interesting. Workload-based analysis could provide several insights in this respect, in addition to shedding light on common usage patterns and how they deviate from the assumptions made by file system designers.

References

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] S. Bailey and T. Talpey. The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols. RFC 4296 (Informational), December 2005.

[3] Peter J. Braam. The coda distributed file system. *Linux J.*, 1998(50es):6, 1998.

[4] Jose Carlos Brustoloni. Interoperation of copy avoidance in network and file i/o. In *INFOCOM (2)*, pages 534–542, 1999.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[6] R. Card, T. T'so, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[7] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[8] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[9] SNIA CIFS Technical Work Group.

[10] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[11] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, 1990.

[12] J. Liang, A. Bohra, H. Zhang, S. Ganguly, and R. Izmailov.

[13] C. Lin. Build object-based filesystem into linux, 2002.

[14] R. Lachaize M. Flouris and A. Bilas. Shared and flexible block i/o for cluster-based storage, 2006.

[15] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system, 2001.

[16] Kostas Magoutis. The optimistic direct access file system: Design and network interface support.

[17] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage, 2003.

[18] Sun Microsystems. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989.

[19] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[20] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). *SIGMOD Rec.*, 17(3):109–116, 1988.

[21] Edward W. Felten Robert A. Shillner. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Princeton University CS Department, 1996.

[22] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. pages 41–54.

[23] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[24] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[25] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[26] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 22–26 1996.

[27] F. Wang, S. Brandt, E. Miller, and D. Long. Obfs: A file system for object-based storage devices, 2003.

[28] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, 1993.

[29] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet firewalls (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.