FlexFS: Transparent Resilience for GRID Storage Resources

Konstantinos Tsakalozos
University of Athens
Athens, 15748, Greece
Email: k.tsakalozos@di.uoa.gr

Vassil Kriakov
Polytechnic Institute of NYU
Brooklyn, NY 11201
Email: vassil@cis.poly.edu

Alex Delis
University of Athens
Athens, 15784, Greece
Email: ad@di.uoa.gr

***Abstract*— Existing GRID infrastructures rely on explicit user instructions in order to replicate files for the purposes of resiliency. This human-intensive process is inefficient, error prone and, more importantly, makes file replication in GRIDs a cumbersome task. To address this problem, we introduce *FlexFS* – a fully automated file-system framework that seamlessly plugs into existing GRID structures providing automated file replication and transparent-to-user resilience. *FlexFS* breaks apart files into blocks and injects resilient information into these blocks through the use of Forward Erasure Correction codes. *FlexFS* employs a number of methods that facilitate the automated storage and efficient retrieval of the blocks in order to provide I/O throughput similar to that of local hard disks, all in the face of ever-changing utilization and availability of the GRID resources. Compared to currently available GRID replication schemes, *FlexFS* attains 15% to 230% higher throughput, both for reading and writing files.**

I. Introduction

The sustained growth of a distributed storage infrastructure, such as a GRID, is subject to frequent resource failures. Malfunctions of physical hard drives, in conjunction with the occasional limited bandwidth availability and heavy workloads at servers, render the effective sharing of storage resources a critical issue. In general, resource unavailability is not a rare case in computational GRIDs [1], [2]. File replication is the solution GRIDs currently offer in order to increase data integrity and achieve better application performance. Dispersing copies of data to multiple nodes improves the chances for opportunistic processing. Dispatching a job to an under-utilized cluster that is close to a replica eliminates the need for file transfers. Moreover, computational GRIDs are meant to provide consistent, pervasive, dependable and transparent access to computing data storage, information retrieval and application services [3], [4]. However, the current provision for resilient data storage services is not automated. The existing means for attaining data integrity in GRIDs is through explicit user commands. File replication is routinely left as a task to the user who has to select specific nodes in which replicas are hosted [5]. Considering that the performance of GRID-enabled applications heavily depends on the proximity of computing elements to the node(s) storing requisite data, it becomes clear that effective and versatile file placement is an aspect of paramount importance. Today, a user is required to be well-versed in the topology of the GRID-network as well as the capabilities of individual computing nodes – both aspects of dynamic nature. In this context, GRIDs currently fail to provide dependable, pervasive and, more importantly, transparent storage services.

In this paper, we overcome the above shortcomings for GRID storage resources by splitting files into smaller data segments termed blocks. We enrich the blocks with redundant information through the use of Forward Erasure Correction codes (FEC), generating packets. Subsequently, we manage the packet allocation to GRID nodes in a manner that is transparent to the user. At the same time, the automated packet dispersal fully utilizes the available resources, while maintaining fast retrieval performance. The packets in question become the core mechanism for the dynamic reconstruction of blocks and files. File reconstruction is accomplished by simultaneously fetching packets from multiple storage nodes that may be either local or remote to a cluster. Even when storage nodes become unavailable, FEC still enables us to recreate the blocks of a file. To reconstruct a block, the number of retrieved packets is such that their cumulative size is equal to the size of the block in question.

We incorporate the above techniques into a GRID-based file-system framework called *FlexFS*, which manages the breaking up of file blocks into packets and handles files transparently to the user in a manner that most efficiently utilizes the available distributed resources. Overall, our proposal offers:

1) mechanisms for an entirely user-transparent file management in computational GRIDs,
2) scalable resilience by efficiently allocating FEC-encoded packets to available nodes,
3) orders of magnitude increase in mean time to failure compared to currently available replication methods,
4) increase of up to 230% throughput performance through efficient packet retrieval methods, and
5) a *FlexFS* implementation that plugs into existing GRIDs.

Two types of nodes comprise the functionality of *FlexFS*: Coordination Nodes (*CNs*) and Storage Nodes (*SNs*). *CNs* manage the metadata necessary to orchestrate the packet dissemination and retrieval, while *SNs* are only responsible for reading data from and writing data to disks. Virtually all long-term memory devices hosting file-systems exported through a *POSIX* API can be used to store packets. *CNs* initially break files into blocks, augment each block with a degree of redundancy, and efficiently allocate packets of such augmented blocks into multiple *SNs*. When reconstructing a file, multi-threading is employed for the simultaneous packet retrieval from multiple *SNs*, terminating requests as soon as sufficient number of packets become available. The techniques used by *FlexFS* to manage packet placement and retrieval allow it to attain much higher read/write throughput rates than existing GRID replication schemes.

Since redundancy is built into each packet, it does not matter which specific packets are retrieved. For example, a *512* KB block can be encoded into four *256* KB packets, each stored in a separate *SN*. Any two out of the four packets are necessary to reconstruct the original block. Consequently, a *CN* node may issue four requests but can nevertheless recreate the block from the first two packets delivered. This method also improves write throughput – instead of writing one *512* KB block to a single *SN*, four *SN*s write *256* KB in parallel, potentially completing the task in half the time. With respect to resilience in this example, as many as two out of the four *SNs* may fail without "loosing" the original block. This is akin to a RAID-6 disk system.

We provide an implementation of *FlexFS* based on the *FUSE* framework [6]. Our prototype integrates into a GRID by exposing the GRID Storage Element (*SE*) functionality through the *Storage*

Resource Management (SRM) API [7], [8]. We evaluate our *FlexFS* prototype in realistic scenarios and identify FEC-configurations that deliver throughput similar to that of a local hard-disk. Simulations show that our approach delivers up to 230% higher throughput compared to existing GRID-replica solutions.

The rest of this paper is organized as follows: Section II presents related work. Section III gives an overview of the current GRID methods for file replication. An introduction to FEC codes is provided in Section IV. Section V presents the implementation details of our framework. Section VI details our empirical evaluations. Finally, our conclusions are presented in Section VII.

II. Related Work

FlexFS adopts Forward Erasure Correction codes (FEC) in order to provide for resilient file storage in the GRID. FEC is a class of error correcting codes that has been used in telecommunications [9] and in computer systems [10], [11] to regenerate packets lost during transmission. The introduction of FEC-encoded data implies processing overheads at nodes while at the same time it offers significant advantages over file replication [12]. In [13], *Tornado* erasure codes are used to improve the speed of information retrieval from mirrored sites. We note that while *Tornado* codes provide faster message encoding and decoding than FECs, they require more packets per block, increasing their utility in environments that emphasize speed over fault tolerance. *OceanStore* uses erasure codes to provide resilience of static files, effectively providing a versioning scheme for file updates through its deep archival storage option [14]. Similarly, erasure codes is an option for adding fault tolerance in the Ursa Minor [15] storage system. In the context of the GRID, there have been application level attempts to use erasure codes in order to provide increased data availability [16]. A widely used alternative to erasure codes is replication. A number of storage solutions proposed use replication since management of replicas is particularly straight forward. GoogleFS [17], FarSite [18], Kosha [19] and the GRID targeting LegionFS [20] all employ replicas to enhance data availability, load balancing and even fault tolerance. In all of the mentioned storage solutions, with the exception of Kosha [19], file metadata as well as directory hierarchy is maintained in specific nodes, thus forming a centralized architecture. On the other hand, Kosha is build on top of P2P networks and since there is no central metadata repository, file and directory manipulation is inefficient.

Our proposal combines some of the characteristics of the systems mentioned above. Contrary to [19], we propose a centralized architectural model where the central nodes achieve high availability and fault tolerance through replication like [17], [18], [20]. However, in the storage nodes (*SNs*), increased data throughput is provided through the use of erasure codes as in the case of [15], [14], [16]. In addition, forward erasure codes allow *FlexFS* to enhance fault tolerance and load balancing at the file block level. Efficient maintenance of directory tree and file metadata is achieved either by having replicas of a central metadata node (*CNs*) or by delegating it to other GRID services that provide the required features. To the best of our knowledge, this is the first proposal for increased data availability in a GRID environment that is transparent to the user.

III. Current GRID Replica Schemes

Current GRID infrastructures are constructed around the notion of Virtual Organizations (*VO*). Each *VO* is a collection of resources that are provided to registered users. The resources fall into two categories: computational and storage-related. Scattered throughout the Internet, a number of physical machines export such resources by hosting specific GRID-layer services. Services offered by the same provider form a GRID site. In essence, each *VO* is an aggregation of such GRID sites. Figure 1 depicts a deployment draft of the most important services present in a *VO*.

GRID computational resources are accessed through a *Computing Element* (*CE*) and tasks are executed on *Worker Nodes* (*WNs*). *WNs* have to access files stored in *Storage Elements* (*SE*) during job execution. The *SE* where each file is located is provided by the *File Catalog* (*FC*). The *FC* functions as a directory for mapping files to one or more *SEs*. To execute a job, a user submits a request to the *Workload Management System (WMS)* [21] which dispatches the job to the best possible *CE*.

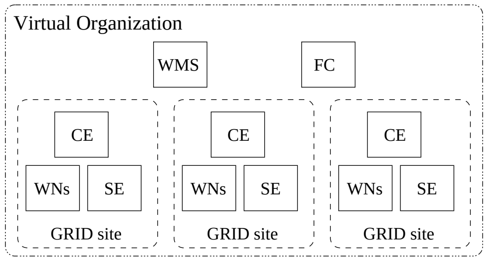

Fig. 1. Basic services of a *VO* in a GRID

To reduce the execution time and resource consumption for a given job, the *WMS* will attempt to place the job for execution with a *CE* that is "closest" to the *SE* holding any required files. How well this proximity criterion is fulfilled depends on the availability of services at the time of job submission. In order to increase data availability, a user can explicitly add file replicas to multiple *SEs* ahead of the computation in order to increase the probability that the job will be allocated to a less-loaded *CE*. Often, this is accomplished by the user issuing explicit *User Interface (UI)* commands to specify the *SEs* in which the files should be replicated. This replication scheme not only bypasses the abstraction layer that GRID aims to provide, but also sets undesired requirements for the users. In this context, users are required to have a good understanding of the *VO*'s GRID sites and the resource each one provides. Therefore, the user must have in-depth knowledge of the topology and capabilities of each GRID site of interest. For instance, if a user intends to execute a job that accesses a file stored in the *SE* of site A, but the availability of the corresponding *CE* is limited, there are two options: either take no action and let the *WMS* dispatch the job to another *CE* sustaining the overhead of transferring the file during job execution, or replicate the file from A to another site. The first option results in undesired delays or even data unavailability in case of limited network bandwidth between GRID sites. In the manual replication option, the user has to determine a new GRID site *B* whose *SE* will host the replica. This decision has to be based on the capabilities of the *CE* of site *B* as the *SE* should be coupled with an appropriate *CE* in order to prevent undesirable file transfers during execution. Evidently, the user should be aware of the available *CEs*, their capabilities as well as the intensity of the workloads that such candidate nodes handle. Although such information is available, it requires sophistication on behalf of users as they have to know the API to query each site and correctly interpret the query results. This is a diversion from the realm of interest of the average GRID user. Even worse, requiring such in-depth knowledge of the topology and capabilities of each site renders a GRID to be a collection of clusters out of which the user indirectly picks one site to execute a job, rather than being the intended resource sharing platform [3], [4].

IV. Forward Erasure Correction Codes

In this section, we briefly introduce the Forward Erasure Correction codes (FECs) that provide the file block resilience in the proposed *FlexFS* framework. In a communication involving FECs, a message m, initially comprised of k packets, is injected with redundant information and encoded into n packets. During this transformation, the packet size remains constant but $n>k$. A *(n,k)* erasure code adds $(n\text{-}k)$\**sizeof(packet)* bytes of redundant information in such a way that the initial message can be reconstructed from *any* subset of k out of the n packets. When used in telecommunication systems to avoid information loss, redundancy is probabilistically calibrated according to expected data loss during transmission. Noisy channels call for higher levels of redundancy requiring messages to be encoded into more packets. In *FlexFS*, the redundancy injected is expressed as a percentage of the amount of packets initially comprising the message m:

$$Redundancy = \frac{n-k}{k} \tag{1}$$

Our framework uses a subclass of erasure codes called *linear* erasure codes. In this subclass the message to be encoded is represented as $\overrightarrow{m} = [m_1 m_2 \ldots m_i \ldots m_k]^T$ where m_i is the i_{th} packet. With the use of a $n \times k$ matrix G, message $\overrightarrow{m}$ is encoded into $\overrightarrow{t} = [t_1 t_2 \ldots t_i \ldots t_n]^T$:

$$G\overrightarrow{m} = \overrightarrow{t} \tag{2}$$

where $\overrightarrow{t}$ is the vector of packets that will be transmitted through the communication channel.

Each row of matrix G, along with vector $\overrightarrow{m}$, define a single packet/element of vector $\overrightarrow{t}$ through an equation of the following form:

$$g_{i,1} * m_1 + g_{i,2} * m_2 + ... + g_{i,k} * m_k = t_i \ \ \forall i \in [1, n] \tag{3}$$

with $g_{i,j}$ being the element of G at row i and column j. In environments where erasure codes are used to deal with packet loss, the value of n is determined based on the expected data loss governed by interference criteria, with the outlook that the receiver will always obtain at least k packets of the transmitted vector $\overrightarrow{t}$. In our proposal, however, n is determined by the amount or redundancy that we wish to add to each packet, as detailed in Section V.

Matrix G is used by the transmitter to encode message m into vector $\overrightarrow{t}$. The receiver uses the same matrix G to decode m only from the subset of packets of vector $\overrightarrow{t}$ which are received. This decoding requires at least k out of the n packets/elements of $\overrightarrow{t}$ and also the corresponding k rows of G as only these are enough to form a linear system of at least k equations as per Eqn. 3. We denote the vector of k received packets as $\overrightarrow{t'}$, while G' contains only those rows of G that correspond to the packets which were received. This linear system is denoted as

$$G'\overrightarrow{m} = \overrightarrow{t'} \tag{4}$$

Solving this system results into revealing message m and requires finding G'^{-1}:

$$\overrightarrow{m} = G'^{-1}\overrightarrow{t'} \tag{5}$$

Finding a solution may present a significant overhead due to two factors: a) The elements of $\overrightarrow{m}$ and $\overrightarrow{t'}$ can be of several *KB* long and therefore require many CPU cycles to be processed. b) High precision arithmetic is also required since even a single bit error can affect the entire message [9]. To avoid the semantic difficulties of sharing matrix G between the transmitter and receiver, it is common practice to pack row i of G along with packet t_i.

V. *FlexFS* Overview and Design Aspects

The primary objectives for the design of *FlexFS* are the provision of file resilience in the face of failing GRID storage nodes and high bandwidth utilization through simultaneous in-network I/O-requests. The framework bases its operation on two types of nodes: *Coordination Nodes (CNs)* and *Storage Nodes (SNs)*. These two nodes are expected to replace the *SE* currently deployed in GRID-sites. By separating the managerial aspect of GRID storage from the data repositories, we permit the Coordination Nodes to contact Storage Nodes of different sites. To allow such interaction, metadata have to be shared among all *CNs*.

Without deviating from conventional GRID design principles, *FlexFS* has to become aware of all available storage resources in its realm. In particular, the *CNs* have to be properly configured in order to use and cooperate with respective *Storage Nodes*. The only requirement that our framework imposes is that *SNs* expose their file-systems in a *Unix POSIX* compliant manner. In this regard, *SNs* can be mounted on local file-systems where *CN*-services reside. As a result, a great variety of storage resources and/or devices can be supported. Network file-systems such as the NFS [22] and AFS [23] can be put into use in our approach. Moreover, our proposal allows for even easier and more flexible solutions when it comes to configuring *SNs*. For instance, one can access a storage node through the use of a single *ssh* account on a *SN* and the *sshfs* file-system on the *CN*.

As Figure 2 shows, a *FlexFS* file-system may present either a standard Unix-*POSIX* API or a GRID-compliant one. Therefore, *FlexFS CNs*, with the addition of a few extra metadata management services, can also be mounted in a local file-system through the *FUSE* framework [6]. As a result, a *FlexFS* file-system can serve as a *SN* to other *FlexFS* file-systems. This property enables the clustering of *SNs* with similar characteristics such as the degree of resilience of individual nodes within the group, geographic proximity of nodes, and aggregate throughput. Figure 2 depicts the layers of our framework and the protocols used for communication.

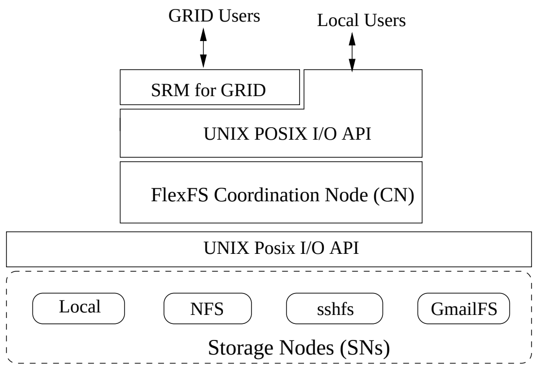

Fig. 2. Layered overview of *FlexFS*

Our relaxed requirements for *SNs* intend to give GRID users an opportunity to readily share storage resources as opposed to the existing mechanisms in computational GRIDs [24], [25]. Using *FlexFS*, and with practically no specific system configuration in place, anyone can contact the *VO* managers in order to offer storage through any (*POSIX* compliant) protocol. It is then up to the GRID site administrators to make use of the offered *SNs* by appropriately configuring the corresponding *CNs*.

The operation of *FlexFS* is centered around the functionality of its *CN* elements. It is there that forward erasure correction is employed and file blocks are augmented with redundant information. Figure 3 depicts the splitting of a file into blocks and shows how blocks are fragmented into packets providing a degree of redundancy. The *degree of redundancy* is a percentage measure of the amount of redundant information encoded into packets. As per Section IV, the degree or redundancy is defined as $100\% \times (n-k)/k$. Packets are finally dispersed among the available and/or preferred *SNs* in proportions defined by the GRID administrator. In this manner, *FlexFS* can put into use *SNs* with different characteristics and capacities. The system administrator has to decide on the desired properties of the *SNs* and select those that better and more efficiently serve the needs of the GRID community. The choice of key parameters such as the redundancy percentage, the block and packet size are based on: (a) the probability of failure of a particular *SN* and (b) the minimum time

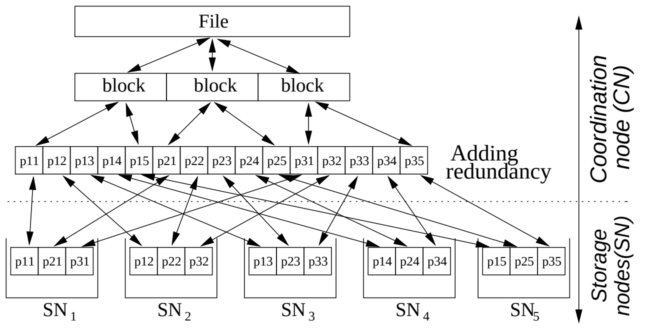

Fig. 3. Files are divided into blocks and then into packets. Packets are ultimately stored at Storage Nodes (*SN*s) distributed in the GRID.

required for a file block reconstruction. In respect to (b), we note that higher redundancy and packet granularity result in the production of more packets. This, in turn, increases the available data sources for block reassembly, effectively reducing the data block reconstruction time. However, as shown in Section VI, there are CPU performance issues that do not allow for unlimited increase of redundancy and number of packets per block.

To reconstruct an entire file, *FlexFS* needs only a subset of the packets distributed to the GRID-network. For each block, the framework will have to retrieve packets whose cumulative size is equal to the block's size. This allows for transparent to the user selection of *SNs* based on their responsiveness and throughput.

A. The Internals of Coordination Nodes

Figure 4 depicts the components that constitute a *Coordination Node* and shows the interactions that take place among them. The entry point of all system calls placed against *FlexFS* is the *top level implementation* which implements the API defined by the *FUSE* framework. There is a one-to-one correspondence between the *Unix I/O* API and the implemented *FlexFS* functions. A number of calls[1] are handled by this very module and do not have additional interactions with other *CN*-modules. These calls work only with the metadata available within the *top level implementation* component. On the other hand, read and write calls are realized in synergy with other modules of the *CN*. During these I/O calls, data are always managed in blocks whose size is different from that used in the local operating system level. Therefore, even if a portion of a block is

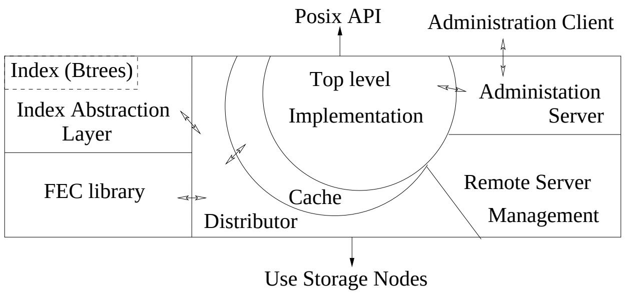

Fig. 4. Internal modules of a *CN*

accessed, the entire block has to be obtained. Assembling each block requires communication with *SNs* which can be time consuming. For this reason the use of a *Cache* is imperative.

[1]such as *stat*, which returns file metadata regarding file size, permissions and access times

The *Cache* module stores decoded blocks in memory so that they can be reused without the overheads of contacting the *SNs* and reconstructing the blocks from packets. Our design of the *Cache* prioritizes read over write requests. Data block writing is carried out in two stages: initially, blocks are placed in a queue in the local file system and during idle periods of the *CN* node they are encoded into packets and the packets are then dispatched to designated *SNs*. While a file has its blocks pending final processing at a queue, the file in question is only available through the *Coordination Node* which has been assigned the task to encode blocks and store packets to designated *SNs*. As soon as this packet "uploading" to *SNs* completes, the data blocks are available at all *CNs*. Therefore, each *CN* must register with the GRID *File Catalog* service that it holds a replica of the entire file. When a file is modified, only the altered/dirty data-blocks are re-encoded into packets, and redistributed to *SNs*.

The *distributor* module commences its work when there are either issued read requests or file blocks are waiting at the queue to be fragmented into packets and stored to *SNs*. Figure 4 depicts the interaction of *distributor* with the other modules of a *CN* node. Should a block be stored, the following steps have to take place: firstly, redundancy has to be added to the block. The percentage of added redundancy is a function of how pervasive in the GRID-infrastructure the file in discussion should be, as decided by the GRID administrator. Evidently, as the number of *SNs* involved increases, the reliability of the file is enhanced as well. Secondly, the block along with its redundant information is fragmented to equal-size packets that are dispatched for storage according to the *FlexFS* placement policy. We use an indexing mechanism so that we can readily track, retrieve and/or update packets of blocks in the system. This mechanism is appropriately updated so that we know at all times where packets of specific blocks of a particular file are stored. In coordinated action with the index, the *distributor* has to follow the reverse approach to assemble a block out of the constituent packets when a read takes place. The multi-threaded design of the *distributor* enables *FlexFS* to be fast by ensuring that many packets are simultaneously retrieved until the desired population of a block is reached.

The *Forward Erasure Correction (FEC)* library provides *FlexFS* with the necessary substrate to overcome either *SNs* failures or non-responding nodes. When writing a file block, *FEC* uses as input the block, the amount of redundancy to be added and the number of packets to be produced. For instance, if we intend to store a file block of $1\,\mathrm{MB}$ with 100% redundancy and use packets of $256\,\mathrm{KB}$, then *FEC* produces eight packets. When reading file blocks and once the work of the *distributor* has completed, the packets are input into *FEC* which produces the block. In the above example, we need to acquire only four of the stored packets ($4 \times 256KB = 1MB$).

B. Metadata Management

The storage of data to remote nodes calls for efficient handling of metadata pertinent to packets, blocks and files. We identify two types of metadata: *a)* metadata necessary to implement a *POSIX* I/O API and *b)* metadata related to file, block and packet placement.

Most of the metadata belonging to the first type are the ones returned by a *stat* system call. We need to preserve such metadata since *FlexFS* presents a standard Unix API through the use of the *FUSE* framework. However, for the specific needs of contemporary computing GRIDs, only a portion of this API is needed [7]. To support this, we store the metadata in a single node accessible by all *CNs*. This decision is in sync with the current architecture of the GRID. As the *FC* GRID service manages the metadata for the files stored in a VO, metadata of this first type are stored there. In essence, these metadata are used to provide a hierarchical directory view of all files.

The second set of metadata is *FlexFS* specific and its main objective is to furnish information to *CNs* so that the latter can locate the appropriate packets. More specifically, upon an I/O request, the appropriate block has to be assembled in the *CN*'s *cache*. As the individual data *Block Size* of a file is known, given the *File Offset* in which the I/O operation has to take place, the *Sequence ID* of the data block is computed: *Sequence ID=File Offset/Block Size*. Using this *Sequence ID*, all pertinent packets can be located with the help of the index. The *distributor* translates the *SN-ID* to the mount point location that a particular *SN* is attached on and concatenates it with the path of the packet in order to retrieve the packet content. Table I shows all *FlexFS*-specific metadata required for files, data-blocks and packets.

File Matadata	Block Metadata	Packet Metadata
Block size	Packet size List of packets Checksum	Storage Node ID Path to packet Checksum

TABLE I
FlexFS METADATA

In order to address all metadata requirements, we propose and experiment with two indexing approaches: the first involves a single-manager indexing-node built on a B+tree access method and the second, a purely distributed one, based on the Berkeley-DB framework [26]. Figures 5 and 6 present the deployment in a GRID *VO* of the single-manager B+tree and *Berkeley-DB* indexes respectively.

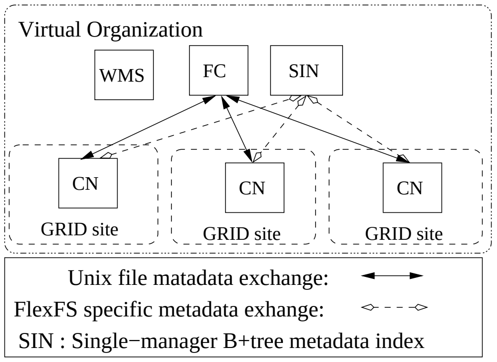
Virtual Organization
WMS
FC
SIN
CN
CN
CN
GRID site
GRID site
GRID site
Unix file matadata exchange:
FlexFS specific metadata exhange:
SIN : Single−manager B+tree metadata index

Fig. 5. *VO* services via a Single-manager B+tree Index

The single-manager B+tree utilizes one index-file for each file stored in our file system. When packets for a particular block are requested, the index mechanism accesses the corresponding index structure on disk and retrieves the *SNs* where the packets are stored. All the index-files are stored in a hierarchical directory structure with the path of each file serving as part of the key used to access the content of indexes. In this regard, when we need to fetch a file, we have to initially traverse the logical path on *FlexFS* and subsequently to access the requisite packets. Clearly, the logical path does not need to be part of the index file itself. This yields compact index files and, more importantly, system calls such as moving and deleting entire files do not require traversing the index. Calls that alter the logical path of a file are realized as corresponding changes in the logical path of the index files. Although having a single manager conduct operations on the index delivers the required consistency, such an architectural choice requires transferring parts of the index to the *CNs* during each *FlexFS* I/O. This approach suffers from two drawbacks: firstly, this central node is a single point of failure as the entire *FlexFS* depends on this node to serve metadata and, secondly, the node hosting the index is a potential bottleneck.

The Berkeley DB engine is used in our second indexing mechanism: this mechanism also implements B+trees and is based on the idea of having a master index site and several mirror sites. The master node maintains full control over the index (read and write access) while the mirror nodes are only allowed to read from their local copy of the index. Each time the index is updated, the changes are propagated to all copies. Requesting an update from all mirror indexes before a transaction is committed ensures that each *CN* has a consistent view of the index. Both mirror updates and ACID transactions are provided by the Berkeley DB engine. In addition to consistency, this index mechanism also offers an improved tolerance to node failure. Should the master malfunction, a mirror immediately takes over. Its election is based on the freshness of the mirrored data and a priority sequence agreed during deployment. This safety mechanism is transparent to *FlexFS* as it is handled by the Berkeley DB framework. In *FlexFS*, each system hosting a *CN* also hosts a node of the Berkeley DB index. This type of index organization delivers increased performance due to the fact that reads may be facilitated through local mirrors and also because of the caching mechanism implemented by the Berkeley DB framework.

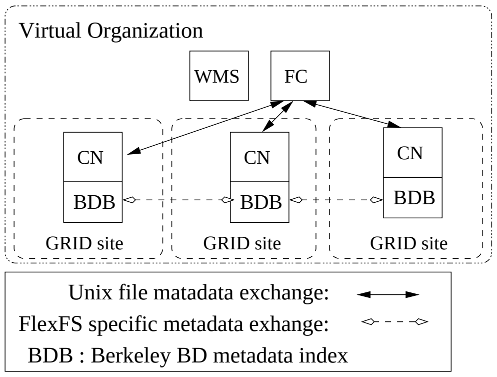
Virtual Organization
WMS
FC
CN
BDB
CN
BDB
CN
BDB
GRID site
GRID site
GRID site
Unix file matadata exchange:
FlexFS specific metadata exhange:
BDB : Berkeley BD metadata index

Fig. 6. *VO* services via Berkeley-DB index

The weak points of the Berkeley DB approach appear when it comes to manipulating entire files and such changes have to be reflected to all mirror sites. For instance, when a file moves to another location in the logical hierarchy, this has to be reflected to all Berkeley DB nodes. Nevertheless, the Berkeley DB solution is far more feature rich than the single-manager B+tree solution. ACID transactions, transparent election of master index node, caching of index blocks and index locking make it an ideal choice for distributed environments. However, such functionality introduces undesirable overheads. Therefore, cases where the single-manager B+tree delivers better performance than the Berkeley DB-based option do exist. We present experimental configurations that expose these overheads in Section VI-B.

VI. Experimental Evaluation

While creating our experimental approach, we set to evaluate three specific objectives: a) to examine the viability of using FEC as a component of *FlexFS*, b) to assess the overall functionality of our *FlexFS* prototype and c) to investigate how our approach compares with the current GRID approach for data resilience.

A. Ascertaining the Feasibility of FECs

In the first series of our experiments, we examine the throughput of the FEC-library that we use in our prototype [27] while varying the following key parameters:

- the block size that is being decoded,
- the packets per block created, and
- the percentage of redundancy that is added.

Our evaluation concentrates on decoding blocks from packets. The process of encoding blocks into packets is not time critical provided that it takes place during the idle periods of the system.

In order to ascertain the feasibility of different CPUs to cope with the requirements that the FEC-library imposes, we use three different computer systems: a) An AMD Athlon(tm) XP 1500+, b) An Intel(R) Pentium(R) 4 CPU at 3.20GHz, c) An Intel(R) Core(TM)2 CPU 6600 at 2.40GHz.

Amount of processed data: We have experimented with a wide range of data block sizes. Here we report the throughput results based on block sizes that are feasible in realistic scenarios: 128KB to 1024KB. We keep the number of packets fixed while varying the block size.

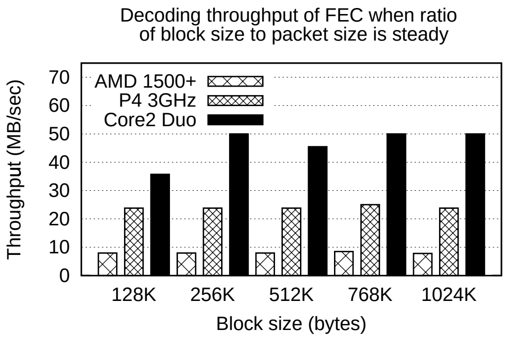

Fig. 7. FEC-Decoding performance with increasing block size and using constant 24 packets per block

Figure 7 presents the decoding performance in one of our tests where the number of packets is set to 24 per block. The FEC decoding process remains unaffected by the volume processed for such a range of modern CPUs. This shows that the block size we choose for *FlexFS* does not hamper performance at the FEC library level. However, this also reveals that using hardware available a number of years ago the throughput performance (8 MB/sec) was far below that of a commodity hard disk. Performance improves with P4 but throughput only reaches acceptable levels on the Core 2 Duo processor. We note that CPU utilization reaches 100% when the library encodes or decodes blocks. This is not an issue since CPUs on GRID SE nodes, which *FlexFS* aims to replace, are not part of the GRID cluster which executes jobs. Therefore, our intention to harvest the CPU on SEs does not hamper the computational resources of a GRID-site. Further improvements in CPU processing power will surely favor *FlexFS*.

Varying the number of packets per block: In this experiment we decrease the bock size from 1024 KB to 128 KB while fixing the packet size at 32 KB and the redundancy percentage at 50%. As a result, the number of packets managed are a function of the block size. Figure 8 presents the decoding throughput performance for our three hardware configurations. Interestingly, this experiment reveals

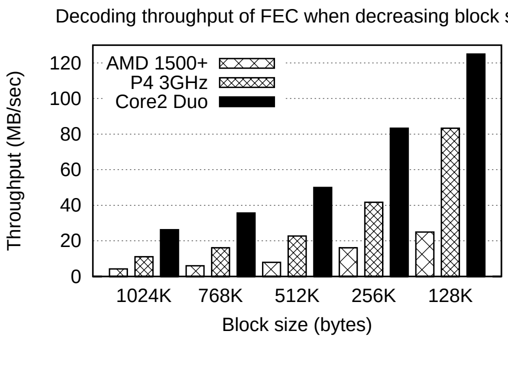

Fig. 8. FEC-Decoding performance when decreasing the block size

a parameter combination that renders the use of FEC inefficient. In most of the configurations the slowest system delivers less than 20MB/sec and it only reaches 25MB/sec when it creates just 6 packets. On the other hand, the Core 2 Duo CPU reaches maximum throughput of 125 MB/sec. Even with a block size of 768 KB (32 packets) the performance of the Core 2 Duo system is acceptable. Taking into consideration these results and the performance capabilities of the hardware where the *CN* services will be hosted, the GRID administrator has to set the number of packets produced for each block. This number is calculated as follows: $NumberOfPackets = (BlockSize + BlockSize * Redundancy)/PacketSize$.

Varying the redundancy percentage: We examine the FEC attained throughput rates when we vary the percentage of redundancy. For this purpose, we set the block and packet sizes to 512 KB and 32 KB respectively and experiment with redundancy percentage from 25% to 200%. Figure 9 shows the derived throughput rates for the three configurations. The outcome clearly shows the trade-off between

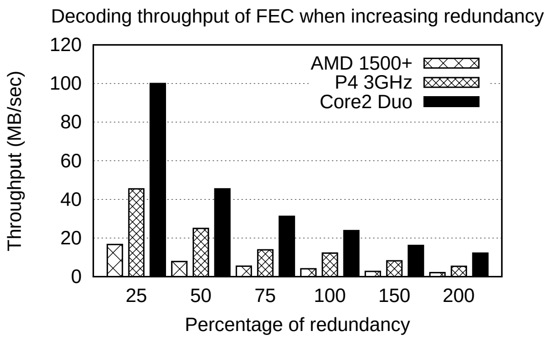

Fig. 9. FEC-Decoding performance when increasing the redundancy percentage

CPU overheads and increased redundancy. Even the fastest CPU cannot cope well when it reaches 150% redundancy and attains only 20 MB/sec. This leads to the conclusion that the use of redundancy rates greater than 100% should be used with caution.

B. Evaluating the FlexFS Prototype

Here we discuss experiments that examine the role of the major components in our prototype and include the size of the B+trees used in both indexing mechanism. We discuss experiments carried out with the AMD Athlon(tm) XP 1500+ equipped with 646MB of main memory. These experiments measure the performance of the *CN*, therefore the testing system should not sustain network delays or display results that are subject to the protocols used between the participating nodes. In order to isolate the *CN* from the *SNs* so as to ensure that it is not the characteristics of *SNs* that dominate the results, we use four storage nodes that are set up as four different directories on the same local ATA-100 5400rpm hard disk drive. Moreover, we do not allow additional *CNs* in order to avoid interference from external processes spawned from other *CNs*. Similarly, the index mechanisms are placed in the same node as the *CN* to eliminate any network delays.

We experiment with the Andrew's [23] and Bonnie++ [28] file-system benchmarks. The main characteristic of Andrew's benchmark is that it involves many small sized files that are repeatedly accessed during an execution. To make sure that I/Os are not served entirely by the *FlexFS* cache module we run multiple instances of Andrew's benchmark at the same time. On the other hand, Bonnie++ benchmark accesses larger files using diverse patterns. The size of the files involved in this benchmark renders it indicative of the performance *FlexFS* will attain in a GRID environments where the file size might reach several GB. For our needs, we take into account the total execution time. In the following tests we start from a base configuration and we gradually change its parameters. The base configuration has index block size: 128 KB, data-block size: 128 KB, packet size: 4 KB, cache size: 50 data-blocks (for a total of 6.4 MB), and redundancy: 50%.

Varying the index block size: We experiment with various index block sizes in the range of 256 KB to 64 KB used by the B+trees that implement both our indexing choices. Figure 10 presents the total execution time for the Andrew's benchmark for three selected block sizes. Figure 11 depicts the corresponding results for the Bonnie++ benchmark. These two figures show that Berkeley DB remains largely

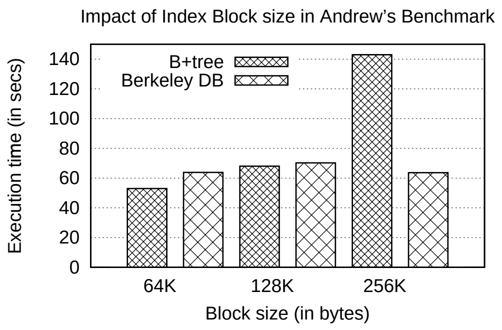

Fig. 10. Andrew's Benchmark with varying index block size.

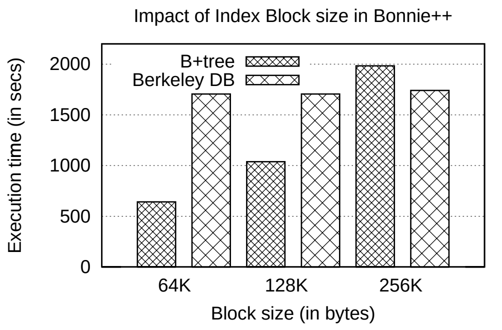

Fig. 11. Bonnie++ with varying index block size.

unaffected by the block size. On the other hand, our single-manager B+tree shows a preference towards small block sizes. The reason for this is that part of the index key in the Berkeley DB index contains the logical path of the file accessed. This is not the case in the single-manager index as it uses a separate index file for each stored file and thus it omits the logical file path from the index keys. The size of the logical file-path is set to 1024 characters. As each index block contains multiple index keys, block size must be several KB so as to compare both index mechanisms with enough keys per index block. In the single-manager index, such large index blocks result in many unused keys fetched through unnecessary hard disk I/Os. On the other hand, block memory mappings and the caching mechanism of the Berkeley DB index pay off. In a similar fashion, for the Bonnie++ Benchmark the single-manager B+tree is also favored by small index block sizes.

C. Using FlexFS in a GRID Environment

In this experiment, we use the *Network Simulator (ns)* [29] to help us assess the effectiveness of the *FlexFS* data resilience approach versus the replication scheme now used by computational GRIDs. The *ns* allows us to set up a controlled GRID environment with six sites connected over the Internet as Figure 12 shows. We assume that the connections among the GRID sites are facilitated through a network of 100 Mb/s interfaces while intranet connections are at 1 Gb/s. In this setting, we also assume that there is one pair of

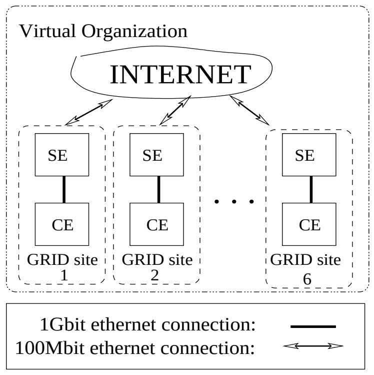

Fig. 12. Simulated GRID deployment.

CE and SE per site. When we experiment with the standard GRID replication scheme, only one storage node (*SN*) participates in the data transfer. On the other hand, *FlexFS* can simultaneously request packets of a file dispersed in a number of *SNs* and may concurrently commence the assembly of the file in question. To this effect, the parameter of primary importance in this experiment is the number of *SNs* taking part in the *FlexFS* operation. Equally critical is the network bandwidth that is available at any time for data transfer from the *SNs* to the *CN*. In this regard, we create random traffic among Internet nodes, reserving from 10% to 90% of the available bandwidth for our data transfers.

Figure 13 presents the results of the GRID simulation when the average bandwidth available for our data transfers decreases. Each line shows the throughput rates achieved by replication and *FlexFS* using between 2 and 5 *SNs*. In relative terms, *FlexFS* does overall

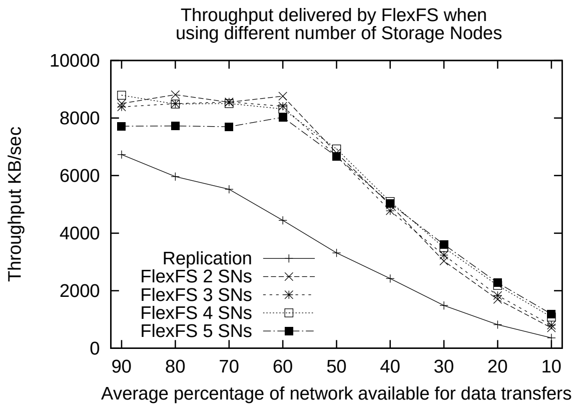

Fig. 13. *FlexFS* throughput rates under diverse network loads

much better than its replication counterpart. When the network is less-loaded there are significant increases in the throughput attained as *FlexFS* achieves between 15% and 31% improvement. More significant are the gains when network bandwidth is constrained (10% on the x-axis) in which case the relative gains for *FlexFS* range between 95% and 229% compared to the GRID replica scheme. In absolute numbers, the above percentiles correspond to a difference of 900 KB/s. However, with *FlexFS* the increase in throughput can reach up to 4 MB/s which is 100% higher than the throughput delivered by the replica scheme. Despite the apparent improvement obtained by *FlexFS*, this experiment also provides some pointers into the number of *SNs* that can be used. When there is no contention for bandwidth in the network, only a few *SNs* are sufficient to take advantage of the network infrastructure. In case of limited network bandwidth the use of more data sources, leads to higher throughput rates.

Our simulations also showed that when the network is less loaded there is a threshold on the optimum number of *SNs* to be contacted. Exceeding this threshold causes low level data packets to be dropped, thus delaying the overall file access. The optimal number of *SNs* is subject to the bandwidth available between network routers, the length of packet queues as well as the network routing algorithms. In the specific setup of the experiments depicted by Figure 13, the threshold, for network utilization above 60%, is four nodes. This is why in a configuration with five *SNs* where more than 60% of the network is utilized by *FlexFS*, the sustained throughput is slightly lower than a two to four SN configuration. In future implementations we will enhance *FlexFS* by setting the appropriate threshold without interrupting its operation.

VII. Conclusions and Future Work

In this paper, we introduce *FlexFS*, a flexible file-system for computational GRIDs and distributed systems. The main objective of *FlexFS* is to automatically overcome node failures that may incapacitate storage servers and may render the current GRID file replication schemes ineffective. We achieve this through injection of redundant information at the data block level, followed by data fragmentation into packets and distribution in the network. The degree of redundancy may vary according to the user needs as well as the potential of storage node unavailability. Forward Erasure Correction is the mechanism we use to create data redundancy. *FlexFS* requires that only a fraction from the original block packets that have been distributed be fetched to cumulatively re-produce a data block sought. We outline the basic functionalities and metadata used by our file-system. *FlexFS* is a viable alternative to contemporary GRID replication as users may work in a entirely transparent fashion regarding the operational state of the network and GRID nodes. Through prototyping and experimentation we demonstrate the feasibility of *FlexFS* and establish the trade-offs for the operation of its key components. Also, through simulations in a controlled environment, we show the performance benefits of the proposed framework over the guarantees that GRID-replication offers. *FlexFS* throughput gains range from 15% to nearly 230%. We are currently working towards the integration of *FlexFS* with the SRM protocol [8] in order to achieve seamless operation in a data-intensive GRID environment. In the future, we plan to investigate how we can dynamically specify the optimum number of *SNs* to be put into use when retrieving data packets. In addition, we plan to investigate policies that automatically match and calibrate the operation of *FlexFS* on heterogeneous sites and examine the *FlexFS* impact on other GRID services.

References

[1] G. Stewart, D. Cameron, G. Cowan, and G. McCance, “Storage and Data Management in EGEE,” in *Proc. of the 5th Australasian Symposium on ACSW Frontiers*, Sydney, AUS, 2007, pp. 69–77.

[2] D. Cameron, R. Carvajal-Schiaffino, P. Millar, C. Nicholson, K. Stockinger, and F. Zini, “Evaluating Scheduling and Replica Optimisation Strategies in OptorSim,” in *Proc. of the 4th International Workshop on Grid Computing*, Washington DC, November 2003.

[3] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1999.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,” The Globus Project, 2002, http://www.globus.org/research/papers/ogsa.pdf.

[5] D. Cameron and et al, “Replica Management in the European DataGrid Project,” *Journal of Grid Computing*, vol. 2, no. 4, pp. 341–351, 2004.

[6] M. Szeredi, “Filesystem in Userspace,” http://fuse.sourceforge.net/.

[7] A. Sim and A. Shoshani, “The Storage Resource Manager Interface Specification,” 2006, http://sdm.lbl.gov/srm-wg/doc/SRM.v2.2.pdf.

[8] Fermilab SRM team, “Storage Resource Manager Project,” https://srm.fnal.gov/twiki/bin/view/SrmProject/WebHome.

[9] L. Rizzo, “Effective Erasure Codes for Reliable Computer Communication Protocols,” *ACM Computer Communication Review*, vol. 27, no. 2, pp. 24–36, April 1997.

[10] P. Barsocchi, A. Gotta, F. Potorti, F. Gonzalez-Castano, F. Gil-Castineira, J. Moreno, and A. Cuevas, “Experimental Results with Forward Erasure Correction and Real Video Streaming in Hybrid Wireless Networks,” *2nd Int. Symposium on Wireless Communication Systems*, pp. 662–666, 2005.

[11] A. Haeberlen, A. Mislove, and P. Druschel, “Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures,” in *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI’05)*, May 2005.

[12] H. Weatherspoon and J. Kubiatowicz, “Erasure Coding Vs. Replication: A Quantitative Comparison,” in *IPTPS*, 2002, pp. 328–338.

[13] J. W. Byers, M. Luby, and M. Mitzenmacher, “Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads,” in *INFOCOM*, 1999, pp. 275–283.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. W., and B. Zhao, “OceanStore: An Architecture for Global-scale Persistent Storage,” in *Proc. of ACM ASPLOS*. ACM, November 2000, pp. 190–201.

[15] M. Abd-El-Malek, I. William V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, “Ursa Minor: Versatile Cluster-based Storage,” in *Proc. of the 4th USENIX Conf. on File and Storage Technologies (FAST’05)*. Berkeley, CA: USENIX Assoc., 2005.

[16] M. Pitkanen, R. Moussa, M. Swany, and T. Niemi, “Erasure Codes for Increasing the Availability of Grid Data Storage,” in *Proc. of the Advanced Int. Conf. on Telecommunications & Int. Conf. on Internet and Web Applications and Services (AICT/ICIW)*, Washington, DC, 2006, p. 185.

[17] S. Ghemawat, H. Gobioff, and S. Leung, “The Google File System,” in *19th ACM Symposium on Operating Systems Principles*, New York, NY, December 2003.

[18] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, “Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment,” in *Proc. of the 5th Symposium on Operating Systems Design and Implementation*. New York, NY: ACM, 2002, pp. 1–14.

[19] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu, “Kosha: A Peer-to-Peer Enhancement for the Network File System,” in *Proc. of 2004 ACM/IEEE Conf. on Supercomputing*. Pittsburgh, PA: IEEE Computer Society, 2004, p. 51.

[20] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, “LegionFS: a Secure and Scalable File System Supporting Cross-domain High-performance Applications,” in *Proc. of 2001 ACM/IEEE Conf. on Supercomputing*. New York, NY: ACM, 2001, pp. 59–59.

[21] G. Avellino and et al., “The First Deployment of Workload Management Services on the EU DataGrid Testbed: Feedback on Design and Implementation,” in *Proc. of Computing in High Energy and Nuclear Physics*, La Jolla, CA, March 2003.

[22] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and Implementation of the Sun Network Filesystem,” in *Proc. Summer 1985 USENIX Conf.*, Portland, OR, 1985, pp. 119–130.

[23] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” *ACM Trans. on Comp. Systems*, vol. 6, no. 1, pp. 51–81, February 1988.

[24] G. A. Cowan, “Disk Pool Manager,” Univ. of Edinburgh, Edinburgh, U.K., Tech. Rep., 2005, http://hepwww.rl.ac.uk/SYSMAN/dec2005/talks/DiskPoolManager.pdf.

[25] M. Ernst, P. Furhmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman, “dCache: a Disrtibuted Storage Data Caching System,” http://www.dcache.org/manuals/talk-4-005.pdf.

[26] M. Olson, K. Bostic, and M. Seltzer, “Berkeley-DB,” in *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[27] L. Rizzo, “Forward Erasure Correction Library,” http://info.iet.unipi.it/~luigi/fec.html.

[28] R. Coker, “Bonnie++ File System Benchmark,” http://www.coker.com.au/bonnie++.

[29] K. Fall and K. Varadhan, “The Network Simulator (*ns*) Manual,” http://www.isi.edu/nsnam/ns/, 2007.