Scalable Computing: Practice and Experience
Volume 10, Number 1, pp. 115–130. http://www.scpe.org
ISSN 1895-1767

THE EDGE NODE FILE SYSTEM: A DISTRIBUTED FILE SYSTEM FOR HIGH PERFORMANCE COMPUTING

KOVENDHAN PONNAVAIKKO* AND JANAKIRAM D*

Abstract. The concept of using Internet edge nodes for High Performance Computing (HPC) applications has gained acceptance in recent times. Many of these HPC applications also have large I/O requirements. Consequently, an edge node file system that efficiently manages the large number of files involved can assist in improving application performance significantly. In this paper, we discuss the design of a Distributed File System (DFS) specialized for HPC applications that exploits the storage, computation and communication capabilities of Internet edge nodes and empowers users with the autonomy to flex file management strategies to suit their needs.

Key words: distributed file systems, high performance computing, peer-to-peer systems, scalability

1. Introduction. Data requirements of High Performance Computing (HPC) applications have been continuously growing over the past few years and are expected to grow even more rapidly in the years to come [23]. There has also been a significant increase in the number of participants, i. e., data producers and consumers, and in their geographical spread. Experimental setups, deployments of sensors, simulators, etc. generate large amounts of data which researchers world over may have use for. I/O libraries that provide necessary abstractions to HPC application programmers require appropriate interfaces and mechanisms for handling the storage, access and transfer of such data. Efficiently sharing large numbers of files among a substantial number of widely distributed users and groups is another related requirement. These requirements create a need for a large scale Distributed File System (DFS) on the Internet edge nodes.

The importance of file management in HPC applications is exemplified by the High Energy Physics (HEP) Data Grid [19]. The project shows the need for collectively using the storage and computing capabilities of widely distributed resources[1] for a specific purpose (in this case, performing next generation HEP experiments). For the HEP Data Grids, there is a need to maintain file catalogs and information services to discover files, locate them (by mapping logical file names to physical addresses) and to provide access. Other examples include Biomedical Informatics Research Network (BIRN) [2], drug discovery on desktop Grids [11], etc.

Traditionally, the set of storage nodes in individual DFS installations comprises of stable and well connected server quality nodes belonging to a few institutional LANs (such as in universities and companies). Advances in hardware technology have made it possible for a much larger set of edge nodes to contribute significantly to the performance of large scale distributed systems. In addition to all the nodes connected to institutional LANs (not just server quality nodes), Internet edge nodes include desktops, laptops, etc. that connect to the Internet using cable, DSL, Wi-Fi, cellular networks, etc. (Fig. 1.1). Our earlier works Vishwa [34] and Virat [40] address issues involved in using edge nodes for providing processor and memory sharing services respectively. In this work, we propose to exploit the resources of Internet edge nodes to build a scalable distributed file system. The presence of a large number of nodes helps in improving I/O scalability, while also boosting the amount of available computational and storage resources. We will, henceforth, refer to our file system as the Edge Node File System (ENFS).

ENFS uses proximity based clustering of edge nodes for the efficient management of resources and balancing of load (storage, computational, query, etc.). A system-wide structured Peer-to-Peer (P2P) network, comprising of a few capable and reliable nodes from each cluster, assists in the efficient discovery and usage of resources across the entire system. Such a two layered platform helps the system to scale to a large number of nodes and also maintain high levels of performance.

Requirements of different HPC applications, in terms of the characteristics and locations of storage sites on which files are stored, vary widely. Access/sharing semantics and consistency needs of users and groups also differ to a great degree. It is, therefore, imperative to consider user-centric preferences while performing file management tasks such as data storage, access and migration. Instead of carrying out such tasks in a rigid manner, in our system, we empower users/applications with the autonomy to flex the system's file management strategies to suit their needs and, hence, improve application performance. For instance, we demonstrate a

*Distributed and Object Systems Lab, Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

[1]The terms resource and node will be used interchangeably in this paper.

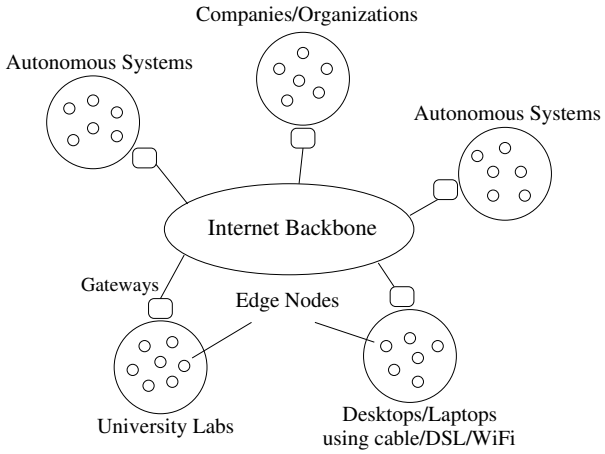

FIG. 1.1. *Internet Edge Nodes*

mechanism that allows a user to continue performing computations on a dataset, while it is being modified by another user.

Important design considerations in the context of a DFS for HPC using edge nodes are discussed in section 2. Section 3 presents the details of our system architecture and the working of ENFS. Section 4 deals with the various aspects of file management in ENFS. Performance studies are discussed in section 5. In section 6, we discuss the features of a few related systems which are relevant to our work. Conclusions and possible future work are presented in section 7.

2. ENFS: Design Considerations. Several features of a distributed file system must be analyzed carefully in order to design one with specific requirements [12]. We will discuss some of the important features in this section in the context of designing a large scale distributed file system that operates in a highly dynamic environment and is specialized for high performance computing applications.

2.1. System Organization. System organization refers to the assignment of specific roles to different system resources and defining how they interact with each other. Distributed file system organization varies from completely centralized [38] to completely P2P [24]. In a completely centralized setup, data and metadata associated with all the files are maintained at a server and clients query the server for all forms of file access. In such an approach, the centralized server severely affects the system's scalability. Making a set of nodes collectively function as data and metadata servers for a much larger number of clients helps in improving scalability to some extent [21]. Pure P2P file systems exist [24], in which every peer makes its own file placement decisions and uses P2P routing techniques to discover files. While such P2P systems scale well, the overheads involved in discovering and accessing files are high as well. Maintaining data and metadata consistency is also expensive in pure P2P systems.

2.1.1. Decoupling Data and Metadata. A recent approach to file system organization has been to decouple file data and metadata [17][44][39][3]. In these systems, only metadata is maintained by a set of servers. File contents are stored on a much larger set of nodes. The total amount of bandwidth that is available to clients to read and write data scales more or less linearly with the number of storage sites available. Such an organization results in a more scalable system.

2.1.2. Metadata Partitioning. A large percentage of file system operations are those on file metadata [36]. Thus, metadata access must be done as efficiently as possible. When a set of nodes function as metadata servers for the entire system, metadata partitioning mechanisms must be used to balance load among the servers. In [44], dynamic subtree partitioning [46] is used to partition the directory space among servers. In addition to namespace management, tasks such as monitoring the status of storage nodes, user/group management, etc. must also be uniformly distributed among a set of nodes.

2.1.3. Data Distribution. It is also necessary to distribute file contents among the storage nodes as evenly as possible to avoid bottlenecks in the system. Systems can either use a probabilistic method or a

deterministic method to distribute data [12]. In the probabilistic method, hashing functions such as Secure Hash Algorithm [15] or Message Digest 5 [35] can be used to determine the sites at which files must be stored [24][29]. Since file placement is randomized, such systems will have little or no control over file locations and over other characteristics of storage sites. In the deterministic method, nodes use whatever knowledge they have of the system to place data. Systems with centralized servers that have global knowledge can easily balance storage load. However, such systems do not scale.

2.1.4. Fault Tolerance. While the abundance of edge nodes helps in improving system performance, higher levels of dynamism in resource characteristics and availability amplify the need for efficient monitoring and fault tolerance mechanisms. Replication of data is a widely used mechanism to tolerate system component failures. A file system employing edge nodes must be adept in replicating data on appropriate resources. Failure detection and migration of data must also be suitably handled. When multiple copies exist, file systems are also responsible for the consistent maintenance of data. Sections 3 and 4 discuss how ENFS addresses the above mentioned issues.

2.2. File Management. The primary function of a file system is to manage the placement and access of files. Due to large data set sizes, recent HPC approaches attempt to schedule computations on resources which contain the required data [32][43], thus reducing the amount of data movement. Therefore, placement of files at appropriate storage sites can significantly improve the performance of applications using the files.

2.2.1. File Placement. High performance computing applications are diverse in terms of hardware and software requirements. Applications perform best when the data that they operate on are stored in resources that suit their purpose best. Some of the characteristics that differentiate applications from each other are computational intensity [28], I/O throughputs [30], inter-task communications, memory requirements, etc. For example, in [4], the authors study the scientific applications GYRO (fusion simulation) [16] and POP (climate modeling) [22]. Various characteristics of the two applications are compared and the authors attempt to identify system architectures that will be most suitable for the execution of the two applications. Using similar application profiling techniques or based on the intuitive understanding of application characteristics, users may be able to specify preferences for the kind of resources on which to store their files.

Users may also be able to specify location preferences for the sites at which their files need to be stored. Based on expected access patterns, users may want files to be stored at or in the proximity of certain nodes or clusters. Certain files may most of the times be used by applications along with a few other files. Such files are best placed in the proximity of the associated files in order to reduce the amount of data transfer during application execution.

2.2.2. File Access. Moreover, access semantics preferred by different applications can vary widely. In a system where mutable data is shared among a large number of users[2], the mechanisms used to manage concurrent access to files can have a huge bearing on the performance of applications requiring those files. A rigid set of policies may not be suitable in such a system. Based on users' access requirements and the current state of file access, the DFS must be able to apply different kinds of access semantics so as to improve overall system performance. Section 4 discusses how file management issues are addressed in ENFS.

2.3. Security and Privacy. Several file systems adopt cryptographic techniques to encrypt their data before storing [24][3]. Doing so ensures privacy since users can now access file contents only if they have appropriate keys. In a few systems, content hashes are stored along with the metadata in order to verify the integrity of retrieved files. There are several security issues associated with large scale distributed systems. Standard practices such as the usage of Kerberos authentication protocols for the servers and clients to authenticate each other and establish secure sessions, using Access Control Lists (ACLs) to restrict the usage of files by different users, etc. are popularly used. While we have not specifically addressed any of these privacy/security issues, we have ensured that the above mentioned techniques can be easily incorporated into ENFS.

3. Architecture of the Edge Node File System. Over the last few years, decoupling metadata operations from read/write operations on files has been established as a strategy that can significantly improve a system's scalability and performance. For extremely large distributed systems (Internet scale) handling billions or trillions of files, a huge number of Metadata Servers (MDS) will be required to handle the load without

[2]Depending on the context, in certain places, the term user may actually refer to the user's application.

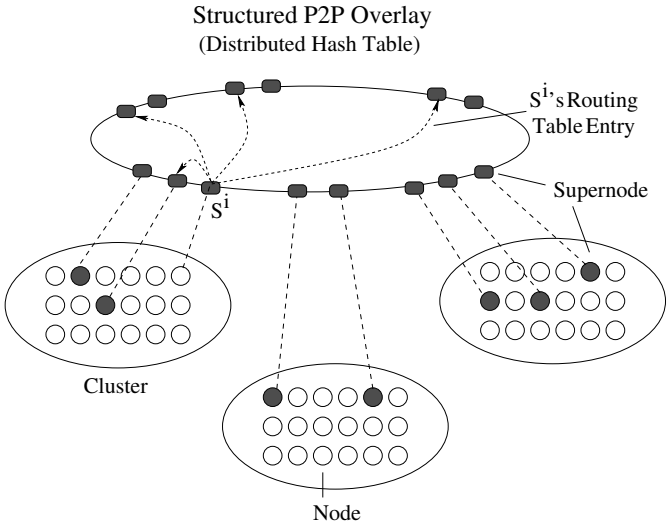

FIG. 3.1. *Two Layered Platform*

effecting any performance degradation. In contrast to many existing systems, in ENFS, we allow non-dedicated edge nodes to assume the responsibilities of a MDS. Such an approach creates a large pool of servers from which the required number can be chosen. Appropriate measures are put in place to handle the consequent issues of failures and load variations.

Metadata (such as file system namespace) has to be partitioned among the MDSs to ensure balanced query loads. Static partitioning of the directory hierarchy is useful in the sense that nodes can easily identify responsible MDSs. However, changes in demand for different portions of the namespace can result in unbalanced loads. Partitioning based on hashing file names (such as in [5]) can help in keeping the load much more balanced. However, such an approach cannot retain hierarchical locality of the files. Moreover, O($logN$) hops may be required to locate the responsible MDS, where N is the number of MDSs in the system. In [46], the issues of changing workloads and retention of locality are attempted to be solved using a dynamically adapting subtree based partitioning strategy. In such a hierarchy traversal system, locating files far down in the hierarchy may require several hops between MDSs. The time taken by a node to identify the MDS responsible for a particular file/directory has considerable bearing on the system performance. In ENFS, we employ a hybrid of hierarchical and hashing based strategies and attempt to minimize MDS location and access times.

3.1. Two Layered System Model. Based on proximity, nodes are grouped into large non-overlapping clusters (order of 10^3 nodes per cluster) by system administrators (Fig. 3.1). New clusters can be formed at any time and nodes can be added to and removed from clusters at any time. A lot of work has been done on network distance measurement [13], topology discovery [18][6] and proximity based node clustering [48][27]. Moreover, statistical data about the number of files, file sizes and generation rates, the number of participating nodes and their characteristics, the number of users and their file access patterns, etc. can be used to determine optimal cluster sizes. In this paper, we do not address issues related to autonomous cluster formation and maintenance for the sake of simplicity.

A few reliable and capable edge nodes from each cluster are chosen to be the metadata servers for that cluster. In addition to maintaining file metadata, information about the nodes within the cluster are also maintained at these servers. We will call these servers *Supernodes*. Supernodes share the load of gathering information about cluster resources, managing users and groups, placing data, maintaining file metadata, scheduling computations, responding to queries, etc. Supernodes from all the clusters form a single system-wide structured P2P overlay network or Distributed Hash Table (DHT) [41][37]. The overlay enables nodes of a cluster to discover supernodes (of other clusters) which are responsible for specific portions of the file namespace. The structured overlay also helps in the efficient discovery of resources with specific characteristics in the entire system.

3.2. Load Balancing. In order to enable efficient distribution of load among supernodes in accordance to their capabilities, a technique similar to the usage of virtual servers in Chord [33] is adopted. In Chord, every

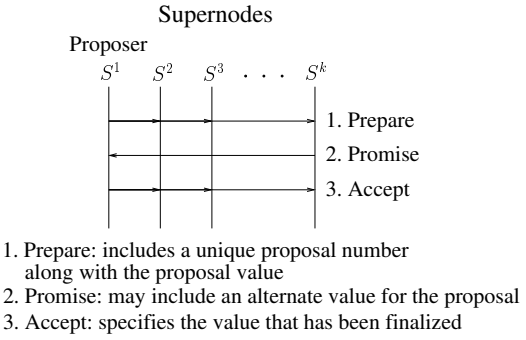

FIG. 3.2. *Paxos Algorithm: Messages and Arguments*

node in the overlay represents not one but a bunch of identifiers from the identifier space. When the query load on a particular node increases, the responsibility for a few of the virtual identifiers is transferred from the heavily loaded node to a lightly loaded node. In ENFS, we distribute a large virtual identifier space among the supernodes belonging to a cluster, based on their capabilities. When the load on a supernode increases, the responsibility for a portion of the virtual identifier space can be transferred from the heavily loaded supernode to a lightly loaded one. We use the Paxos algorithm [25] (discussed later in this section) to unambiguously rearrange load.

The mapping between the virtual identifier space and the physical address of supernodes is made known to the entire system using the structured layer. A file containing the association is stored in the structured layer with the cluster identifier as the key. We will refer to this file as the *supernodes_map*. Supernodes, primarily, manage entities such as nodes, users, groups and files belonging to their cluster. These entities determine the virtual identifier with which they must associate themselves by performing a standard operation on their unique identifiers (IP address, file path, etc.). For instance, let the maximum value of the supernode virtual identifier be V_{max}. The virtual identifier corresponding to an entity with identifier I can be

$(SHA(I)/SHA_{max}) * V_{max}$

where $SHA(I)$ is the digest obtained by applying the SHA hash function on the identifier and SHA_{max} is the maximum possible value of the 160-bit digest. Entities can then use the *supernodes_map* file to determine the physical address of the supernode responsible for that virtual identifier. For an entity with I as its identifier, the responsible supernode is denoted by S_I. When a node wants to make a query about I or when I wants to perform some file system operation, S_I is contacted.

The load at a supernode refers to the amount of computational power spent on performing different file system operations, including responding to client queries. Each supernode specifies a value for the maximum allowed CPU utilization percentage (measured over a time interval) for file system operations. Since file access frequencies and the amount of computational power required for different file system operations vary, loads on the supernodes keep changing. Whenever the actual load exceeds the limit imposed, a supernode is said to be overloaded. Supernodes periodically send their load information to the other supernodes in the cluster. In addition to the sharing of load information, the message also helps in letting other supernodes know about the source's liveness. When a supernode S^i detects the failure of another supernode S^j, it informs the other supernodes in the cluster about the failure[3]. S^i also initiates the process of replacing S^j with a new supernode. A supernode that is overloaded can propose an adjustment of the virtual identifier space to transfer some of its load to lightly loaded supernodes. When all supernodes in a cluster are heavily loaded, an additional supernode can be added to the existing set. On the other hand, when all the supernodes in a cluster are lightly loaded, one of the supernodes can be removed. A minimum number of supernodes, k_r (discussed in section 3.3), however, must always be available within a cluster.

As mentioned earlier, the Paxos algorithm (Fig. 3.2) is used by supernodes to reach an agreement on the addition/removal of a supernode and on the adjustment of the virtual identifier space. In the first phase of the Paxos algorithm, the proposer sends a *prepare* message to the acceptors and the acceptors may or may not respond with *promise* messages. In the second phase, if the proposer receives *promise* messages from a majority

[3]While S^i denotes the i^{th} supernode in the *supernodes_map* file, S_I denotes the supernode associated with entity I.

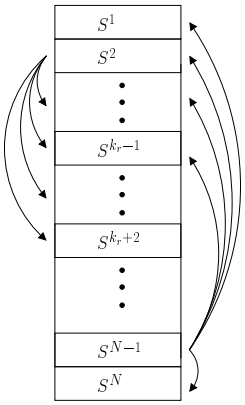

FIG. 3.3. *Supernode Metadata Replication*

of the acceptors, it sends an *accept* message with the decided value to the acceptors. If enough *promise* messages are not received, the proposal fails. Using Paxos helps in achieving consensus among a set of supernodes which are prone to failures. Changes in the set of supernodes and in the mapping of virtual identifiers to supernodes are immediately propagated to the *supernodes_map* file in the structured layer. The file includes a version number which is incremented after every change. The changes are also publicized to all the nodes within the cluster.

3.3. Fault Tolerance. In order to handle failures, the metadata stored in a supernode is replicated in a constant number (k_r) of other supernodes. The number k_r must be selected in such a way that the simultaneous failure of $k_r/2$ supernodes within a cluster is highly improbable. A supernode, S^n, that updates some metadata has to multicast the changes to all its replicas, so that the replicas are always in a state similar to that of S^n. Let there be N supernodes in a cluster and let $\{S^1, S^2, \ldots, S^N\}$ be the order in which the supernodes are listed in the *supernodes_map* file. Then, supernode S^i replicates its metadata in the nodes

$\{S^k, k = (i + j - 1) \bmod N \mid 1 \leq j \leq k_r\}$.

We will refer to this set of supernodes as S^i's replica set. In figure 3.3, all the supernodes pointed at by the arrows originating from supernode S^i constitute its replica set. When a querying resource detects that a supernode is not reachable, it queries the next-in-line supernode in the failed supernode's replica set. Whenever the responsibility for a virtual identifier is transferred from S^i to S^j, appropriate metadata transfer must happen. The metadata of all the resources that map to that virtual identifier must be transferred from S^i to S^j and to S^j's replica set.

Every node, when it becomes a supernode, identifies another node which is capable of being made the supernode, and maintains it as a *spare* supernode. The spare can be made a supernode in case of failures or load surges. Supernodes lazily keep updating metadata information in the spare nodes. This way, the amount of metadata transfer required for bringing up a new supernode is significantly reduced. When a supernode fails, the spare of the failed supernode or that of one of its replica set supernodes can be used as a replacement. Similarly, one of the better prepared spares, i. e., spares with little metadata transfer pending, can be used while adding additional supernodes to a cluster.

3.4. Metadata Consistency. Periodically, a supernode, S^i, sends the collected status information of all the resources it is responsible for to all the other supernodes in the cluster. We do not attempt to achieve sequential consistency among supernodes for such non-critical information because it can be expensive. However, for critical information such as file metadata, S^i and its replica set must always be in a synchronized state, i. e., sequential consistency semantics need to be observed. Otherwise, if S^i performs some file operation, such as providing a write lock to a user, and fails before letting its replica set know about the lock, the available metadata in the replica set becomes inconsistent with the actual state. For similar reasons, user/group metadata must also be maintained consistently among supernodes and their corresponding replica sets.

We adopt the Paxos agreement protocol for the consistent maintenance of critical metadata among S^i and its replica set. S^i sends the proposal to its replica set, waits for a sufficient number of responses, performs the

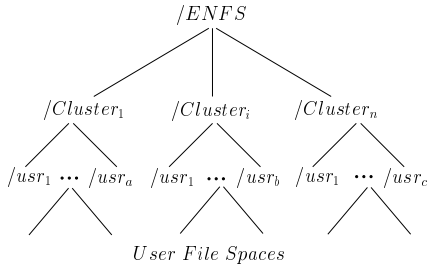

FIG. 3.4. *Hierarchical File System Namespace*

proposed operation and informs the replica set about the completion of the operation. Even if S^i fails before sending the completion message, its replica set can verify if the proposed operation was performed to completion or not. This way, a total order is maintained among the updates made to such metadata. Since the replica set size is a constant, the overhead of maintaining consistency does not increase with an increase in the number of supernodes in a cluster.

3.5. Namespace Management. We employ a single hierarchical file system namespace with the root named as $/ENFS$ (Fig. 3.4). Each filename, including the path, is therefore unique. The first level of namespace partitioning is achieved by making the supernodes of a cluster ($CId1$) operate as metadata servers for all the files and directories belonging to their cluster, i. e., all files and directories having a prefix of $/ENFS/CId1/$. Every user/group in ENFS must belong to one of the clusters. In order to be unique system-wide, user identifiers can be of the form $ClusterId/UserId$ and user home directories can be of the form $/ENFS/ClusterId/UserId/$. Distribution of responsibilities for files and directories among the supernodes in a cluster is done as is discussed in section 3.2. Since filenames including paths are hashed to identify responsible supernodes, metadata of different files in the same directory may be managed by different supernodes. In order to support POSIX directory access semantics, whenever a file or subdirectory is created, updated or deleted, the metadata of the parent directory must also be updated. Symbolic links are supported in ENFS by including the entire name of the original file or directory in the link's metadata. All operations on links are forwarded to the supernodes that manage the linked file's metadata.

Users requiring access to their own files and to the other files belonging to their cluster are served metadata by supernodes of their own cluster. Since cluster formation is based on the proximity of nodes, network latency between clients and metadata servers for such file accesses is small. In order to determine the metadata server (supernode) responsible for any other file, clients will need the current *supernodes_map* file of the cluster to which the required file belongs. This may require O($logN$) hops, where N is the size of the structured overlay. This cost is, however, amortized over future accesses to files belonging to the same clusters. Supernodes of a cluster prefetch and maintain the *supernodes_map* files of all clusters, whose files have been recently accessed by their users, thus thwarting the need to query the structured layer for every single access.

Though hashing helps in balancing query loads to a large extent, sudden increases in demand for individual files, referred to as flash crowds, can still overload the corresponding supernodes. In ENFS, when a supernode, S^i, notices an increase in the number of queries for a particular file, the cluster's *supernodes_map* file is immediately updated with a note suggesting that queries for the popular file must be forwarded through the client's supernode. Supernodes of other clusters that receive such queries, then, fetch and cache the popular file's metadata locally. These supernodes respond to future queries themselves, until the note is removed from the *supernodes_map* file of S^i's cluster (when demand for the file falls). The distribution of the responsibility of serving the metadata of popular files is shown in figure 3.5. Flash crowds can be handled in this manner just for read-only accesses. When the requests from flash crowds are largely update requests, performance is likely to get affected.

Network partitioning within a cluster is taken care of by the active replication and migration of metadata and data across the various supernodes and storage nodes of a cluster. When a small portion of the cluster becomes unreachable, it is unlikely that any data/metadata will become inaccessible.

3.6. Resource Monitoring. Supernodes of a cluster are responsible for monitoring the status of cluster resources. Nodes periodically send their status information to the appropriate supernodes. In addition to

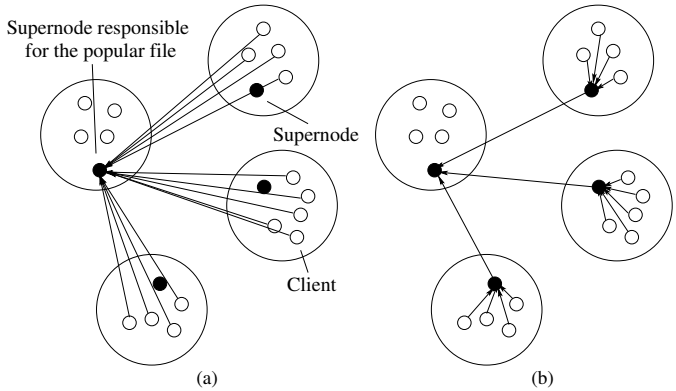

FIG. 3.5. *Handling Flash Crowds (a) Occurrence of flash crowds (b) Distribution of responsibility*

the current values of vital node characteristics, the status information also includes the version number of the *supernodes_map* file that is known to the nodes. If there is mismatch between the actual version number and the received number, the supernode sends the node the current file. This is done to ensure that the nodes in a cluster have current knowledge about the supernodes responsible for that cluster. When a supernode, S^i, does not receive the status information from a node for a certain period of time, it checks if that node has failed. If the node has failed, S^i periodically attempts to ping the failed node. When the node becomes active again, S^i sends the node the current *supernodes_map* file. The frequency and duration of node failures are used to quantify the reliability of a node. This measure is maintained at the supernodes along with other node characteristics. Node status information and failures are immediately made known to the supernodes in S^i's replica set.

3.7. Resource Discovery. The two layered system model of ENFS enables the efficient discovery of system-wide resources with specific characteristics. Since supernodes periodically broadcast status information about the nodes they are responsible for, to the other supernodes in their cluster, all the supernodes of a cluster maintain more or less up-to-date information about all the nodes in their cluster. Supernodes also maintain information about the clusters closest to them in the system. When a supernode is not able to locate enough resources with specific characteristics (to handle a user request) within its cluster, it can send appropriate queries to the supernodes of nearby clusters. However, in certain scenarios, a supernode may not be able to locate enough suitable resources in any of its nearby clusters. Flooding the system with queries loads the network and also does not provide any guarantees. ENFS exploits the reach of the structured overlay to efficiently discover resources located anywhere in the system.

A node is usually characterized by different capabilities or features such as processing speed, memory size, storage capacity, uplink/downlink bandwidths, system architecture, etc. Each characteristic c_i can be quantified by values belonging to m_i categories. For example, memory sizes can be categorized as being less than mr_1 MBs, between mr_1 and mr_2 MBs, ..., greater than mr_m MBs, etc. Standard system-wide policies can be used for the discretization of the values of the different characteristics. For every category of each characteristic, a file is maintained in the structured layer that includes the identifiers of all the clusters that have a large number of resources of that capability available for use. The file also includes the approximate number of resources in each cluster. JuxMem [5] uses a similar strategy for the available memory alone.

The *characteristic_category* file in the structured layer must be updated by supernodes only when their clusters have a large number of resources (with the corresponding capability) on a regular basis. A pessimistic estimate must be used for the number of resources listed as available. This is important because every small variation in the number of resources available in a cluster should not necessitate a change to the file in the structured layer. Using such an approach, the number of clusters to which queries must be sent for resource discovery is significantly reduced. Discovering resources with a combination of characteristics is also possible by making individual queries for each characteristic and then joining the results.

4. File Management. A useful, yet ignored feature in distributed file systems, is that of allowing users to have some level of control over choosing appropriate resources for the storage of their files. Most of the times,

users generating data files, to be used by high performance computing applications, have some knowledge about the characteristics and requirements of the applications that are going to use their data. Information about the set of users who are going to perform computations on the files may also be sometimes available. Data placement based on such knowledge can result in significant improvements in application performance.

4.1. File Placement.

4.1.1. User Preferences. As discussed in section 2.2.1, high performance computing applications differ widely in their characteristics, and hence, resource requirements. Resource requirements can be specified in terms of the processing speed, network bandwidth, memory size, storage capacity, architecture, etc. Location preferences, on the other hand, can be specified based on expected access patterns. Users may want files to be stored in the proximity of specific nodes or clusters. For example, if certain files are expected to be used by the members of a particular group, the files are best placed in the proximity of the nodes frequented by those group members. Certain files may most of the times be used by applications together with a few other files. Such files are best placed in the proximity of the associated files in order to reduce the amount of data transfer required during the execution of the application.

To register preferences about resource characteristics, users must specify values for the different characteristics that storage nodes are expected to possess. The preferred characteristics must be specified as a prioritized list. To indicate location preferences, users can specify the set of users, groups, clusters, files or directories in the proximity of which their files must be placed. As in the previous case, location preferences must also be specified in the order of favor. Replication of user data is necessary to distribute the load on storage sites, handle storage site failures and also to enable parallel access. Users can explicitly specify the required levels of replication for their files based on expected demands.

The following shell command shows how a user can load a file into ENFS:

$ enfs put [options] *local_filename enfs_filename*

User preferences can be specified using command line options, as is done in POSIX commands. For example an option such as –cpu=5 can be used to specify that the file must be stored on nodes with processing speeds greater than 5 GHz. It is not practical to specify several options for each file. The required options can be set at the level of a directory and all the files created in that directory can inherit the set options. On the other hand, users can create combinations of options that they use often (locally on their nodes) and just specify the name of the combination while loading files. The DFS component on the node can replace the name of the combination with the entire set of options.

4.1.2. Block Management. In ENFS, files are split into data blocks. As much as possible, the blocks are stored on different nodes. Among other benefits, this provides for parallel access and fault tolerance. The size of a block can be configured by the user based on application requirements. By default, the system sets the block size in accordance to the file size. ENFS uses the native file system present in a node's Operating System (OS) for the storage of data blocks as files in the local storage devices. The local file names are stored in the ENFS file's metadata. Block storage in the local file system and block transfers between nodes happen as a binary stream of data. This way, differences in the file storage formats of different file systems cannot corrupt the contents of data blocks. The network subsystem and other hardware interfaces of the OS are used for purposes such as communication, monitoring, etc.

4.1.3. Placement Strategy. Each supernode maintains a database with entries for all the resources in its cluster. Such a database assists in quickly identifying resources with required characteristics. Resources from other clusters can be discovered as discussed in section 3.7. Supernodes keep track of the set of nodes that each user frequents. Aggregating this information from all the members of a group, supernodes can associate a set of nodes with each group. From the metadata of files belonging to a directory, supernodes can determine the set of nodes that store most of the data belonging to files in that directory. All such information is gathered by supernodes lazily (as low priority tasks) and is used to make locality based placement decisions.

When users do not specify any preference for resource characteristics, supernodes select the set of storage nodes considering just the available storage space. When no location preference is indicated, supernodes try to store all the replicas within the same cluster and in nearby clusters. While placing data blocks, supernodes also take into consideration the reliability of nodes. Since files have to be available at all times, at least a few of the replicas must be stored on nodes with high reliability, while the other replicas can be stored on nodes with lesser reliability. At a high level, our file placement algorithm performs the steps shown in figure 4.1.

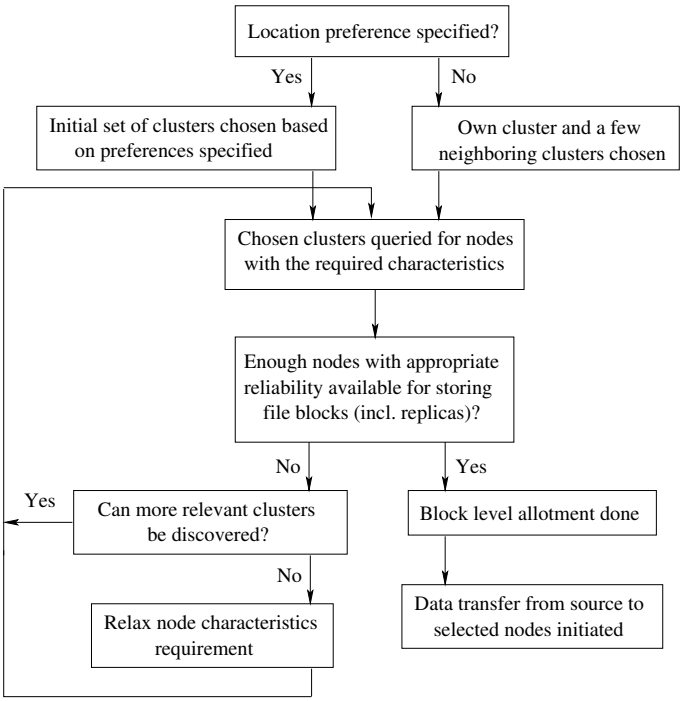

FIG. 4.1. *Bird's Eye View of the File Placement Strategy*

4.2. File Access. In ENFS, file metadata is maintained on reliable and capable nodes, with a single point of access for each file. Such a design makes it convenient for the system to support a large spectrum of access semantics. The single interface allows clients to use the data in various modes. Concurrent accesses can also be easily coordinated. For example, concurrent accesses to files can be controlled by associating files with exclusive and shared locks. All exclusive locks must have been released before shared locks can be obtained by clients and all kinds of locks (shared and exclusive) must be released before an exclusive lock can be obtained. Lock granularity (block level or file level) and lifetime can also be configured. Applications can use ENFS library functions such as *lockf()* and *lockb()* to lock/unlock files and blocks respectively. These functions take as arguments the filename, block identifier, type of lock, lifetime, etc. Supernodes ensure that locks are not violated while responding to read and write requests from multiple users.

4.2.1. Access Semantics. By selecting appropriate access modes, applications can improve their performances significantly. Selecting the right kind of lock can reduce wait times to access data. Due to concurrent accesses, it is possible that multiple versions of a file are available. A user, not too particular about requiring the latest version, can request for read access to any version of a file [42]. This way, the user is more likely to gain access faster. Users can also specify whether the order in which their updates are applied on files is important or not. When the updating order is not important, multiple updates can be simultaneously applied on the various replicas and hence high performance can be achieved. Commutative updates such as counter increments/decrements can be unordered. The *setattr* shell command can be used to modify such options for a file:

$ enfs setattr [options] *enfs_filename*

Caching file contents locally helps in reducing data access times during subsequent usages. In order to ensure that contents have not been modified, files and blocks maintain version numbers (included in the file metadata). Version numbers in the metadata can be compared with version numbers of data items cached by users to determine if cache contents are valid or not. Block level version numbers are useful, especially in the case of large files. Without block level version numbers, caches can be invalidated even when a small portion of a large file is altered.

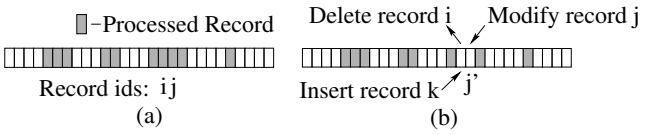

FIG. 4.2. *(a) Progress of a Computation (b) Changes Made to the File*

4.2.2. Computation Scheduling. Most high performance computing applications involve operations such as filtering the contents of a file to extract useful information, enforcing data transformations or performing CPU intensive scientific computations on file contents. Since supernodes have information about all the resources in their cluster and are capable of discovering resources from other clusters, they are well suited for the job of scheduling different kinds of operations on file contents. Based on the specified access preferences, supernodes schedule tasks on appropriate nodes and have the results returned to the user. For example, in order to execute a divisible load application [7] on the contents of a file, supernodes can efficiently set up computation specific dynamic overlay networks, such as the one discussed in our earlier work [31].

In a large scale system involving several users and long running applications, scenarios can arise in which user U_a makes changes to a file while another user U_b is performing a high performance computation on the file contents. A few examples for scenarios where computation results need to be kept up-to-date with changing datasets are web crawlers providing fresh content, re-runs of experiments resulting in better values for a portion of the dataset, sensors detecting new events, etc. Requiring U_b to restart the computation after every change made to the file is not efficient since file sizes are usually huge and computations take a significant amount of time. Similarly, not allowing changes to be made to the file when a computation is being performed can lock the file for long periods of time. In ENFS, we address this issue by enabling the user to provide certain functions (similar to MapReduce [14]) to the responsible supernode which can then handle simultaneous updates to the file while computations are in progress on the data. For certain classes of application, such as embarrassingly parallel applications with commutative updates, the scheduler can continue processing one portion of the file when another portion is being altered, because there is little or no inter-task communication involved. Thus, when changes are made to a file, if it is possible to undo the effects of deleted records from the result, the computation can be made to be consistent with the current contents of the file. For such applications, it may be possible for the user to provide the following details/functions to the supernode:

- The types of *record* and *result*
- The function $compute(record) => result$—represents the computation that is performed on every record.
- The function $aggregate(< result > *) => result$—aggregates a set of results into one.
- The function $recompute(result, record) => result$—reverses the effect of the computation of a single record from the current result.

Let us assume that the computation initiated by U_b has progressed as shown in figure 4.2(a). Each small box represents a record. Shaded boxes represent records that have been processed and unshaded blocks represent records that have not yet been processed. Let us also assume that U_a modifies the file as shown in figure 4.2(b). U_a deletes the record labeled i, inserts record k and modifies record j, the modified record being labeled j'. In order to make U_b's computation consistent with the current contents of the file, the following functions must be invoked by the supernode:

1. $result = recompute(result, i)$
2. $result = recompute(result, j)$
3. $result_1 = compute(k)$
4. $result_2 = compute(j')$
5. $result = aggregate(result, result_1, result_2)$

Steps 1 and 2 are required to undo the effects of old records i and j on the result. Steps 3 and 4 obtain the results of performing the computation on new records k and j'. The final step combines the three results into one. The updates will then have been incorporated into the result and the computation can continue with the records which have not been processed yet. With such user defined functions, ENFS can be customized to support a wide variety of access semantics.

4.3. Data Migration. Data stored in a storage node may have to be migrated to other nodes due to several reasons such as changing access patterns, failure of storage nodes and changes in the characteristics of nodes. Data migration and creation of new replicas based on access patterns help in improving application performance. In ENFS, data transfer time is saved by migrating blocks closer to the nodes/clusters where they are being frequently transferred to. This reduces the effects of high network latency on access times. Moreover, based on the frequency of access, new replicas are autonomously created and unused ones deleted.

A supernode ascertains that a node has abnormally failed only if the node remains offline for extended periods of time. When a supernode detects an abnormal node failure, it passes on the information to the supernodes responsible for the files stored in the failed node. The respective supernodes can then create new replicas. Changes in node characteristics are also handled in a similar manner.

5. Performance Studies. A prototype version of ENFS has been implemented and in this section, we present preliminary performance results. We have used the Andrew Benchmark [20] to assess ENFS. Andrew Benchmark (AB) is widely used to study file system performances and consists of five phases, namely: 1) creation of directories, 2) copying files into the directories, 3) reading file attributes, 4) reading file contents, and 5) performing a hybrid set of operations on the files, such as reading, computing and writing. All the five phases of the workload are relevant in the context of a DFS for HPC applications. Phases 1 and 3 characterize metadata creation and access performance. Phases 2 and 4 characterize data write and read performance. Phase 5 measures the support provided to computation tasks. The performance of a DFS relative to popular file systems such as NFS can be used to make indirect comparisons between itself and any other DFS for which results exist.

Our experimental setup includes a set of 80 machines belonging to two labs in our department. Most of the machines have Pentium 4 processors (3.8 GHz), 1 GB RAM and 80 GB hard disk drives. The predominant operating system is Linux version 4. The machines are all connected to a 100 Mbps LAN, with 30 machines in one subnet and the remaining 50 in another. The machines are split into 2 to 5 clusters, each cluster having between 16 and 70 nodes, depending on the experiment. On an average, each cluster maintains 4 supernodes. The NFS server is a Pentium 4 machine of 3.8 GHz processor speed and 4 GB RAM, running NFSv3.

TABLE 5.1
Andrew Benchmark: ENFS vs. NFSv3

Phase	ENFS (ms)	NFSv3 (ms)
Phase 1	40	8
Phase 2	72066	83227
Phase 3	202	51
Phase 4	34905	86487
Phase 5	3141	28627

TABLE 5.2
Overheads of the Paxos Algorithm

No. of Supernodes	Time taken for Consensus (ms)
2	47
16	62
32	108
64	269

Table 5.1 shows the running times of the different AB phases for ENFS and NFSv3. Phases 2 (write) and 4 (read) involve a lot of data transfer (800 MB in each phase). Since data storage is distributed among several nodes, the running times of these phases are lesser in ENFS than in NFS. ENFS also performs significantly better than NFS in phase 5 because it exploits the computing power of storage nodes to parallelize computations. Overall, in this setup, ENFS performs about forty percent better than NFS. In comparison, Pastis [9] achieves execution times lesser than two times that of NFS, while Ivy [29] and OceanStore [24] perform two to three times worse than NFS.

Multiple instances of AB are started in different nodes simultaneously and the performance studied. Figure 5.1 shows the average execution time per run for the different number of simultaneous AB runs (1 to 5). In ENFS, when the nodes on which AB is started belong to different clusters, the average execution times is more or less a constant. When the nodes belong to the same cluster, a small increase in the execution time is noticed with every additional node. This is because, in the first case, a larger set of supernodes cater to client requests than in the second case, where only a single cluster's supernodes handle all the load. NFS performance deteriorates rapidly with the number of runs because of the single server which serves data and metadata to all the nodes.

The same workload is used to measure the execution times of AB runs in the Hadoop Distributed File System (HDFS) [8]. HDFS is an open source file system with an architecture similar to that of the Google File

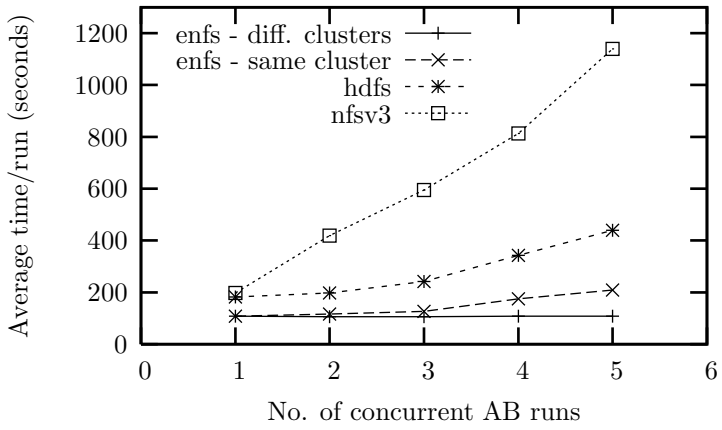

FIG. 5.1. *Concurrent AB Runs*

System (GFS) [17] (discussed in section 6). A HDFS cluster maintains a single *namenode*, which manages the namespace, and several *datanodes*, which store data blocks and serve read and write requests. Since data storage is decentralized, HDFS also performs much better than NFS. However, since the metadata is still managed by a central server in HDFS, its performance is effected when the number of simultaneous file system operations increases. When five concurrent AB runs are started, HDFS's average benchmark time is more than two times that of ENFS.

We measure the overhead introduced by the Paxos algorithm while trying to achieve consensus among supernodes. Table 5.2 shows the average time taken to achieve consensus among varying numbers of supernodes within a cluster. We observe that, though agreement time increases with the number of supernodes, the overheads are not too high (about a quarter of a second for 64 supernodes in a LAN setting).

A flash crowd scenario is created and the response of ENFS is studied. Table 5.4 shows the average response times for metadata requests as observed by clients, with and without load distribution. As expected, the average response time reduces significantly when the responsibility of supplying popular file's metadata is distributed among the supernodes of the clusters from which requests are being issued.

Table 5.3 shows the execution times for a computation on a 600 MB dataset while it is being updated by other users. The time taken to *compute* a single record, of size 1 MB, on a 3.8 GHz Linux machine is 102 ms and the time taken to *recompute* the result when a record is updated is 151 ms. It may be noted that the *recompute* function is applied only on the updated records and not on the entire file. Execution times increase linearly with the rate of updates to the file. Also, for the entire computation, we observe that execution times are lesser for smaller block sizes. As the block size increases, the number of storage nodes used to store the file decreases. In other words, the number of nodes, whose processing power is directly usable for the computation decreases, thus increasing computation times.

TABLE 5.3
Computation Execution Times

Updates /second	Execution Time (seconds) Block size (MBs)		
	20	50	100
4	4	8	15
10	6	11	17
20	10	15	22
40	17	23	31

TABLE 5.4
Flash Crowds: Average Response Times

	Avg. response time (ms)
Flash crowds	29
After load distribution	11

6. Related Work. Certain distributed file system protocols such as *Network File System* (NFS) [38], *Andrew File System* (AFS) [21] and *Common Internet File System* (CIFS) [26] employ the client/server model. File data and metadata are both managed by a single or a limited set of servers. Since the servers are

potential bottlenecks, these systems do not scale with increasing numbers of simultaneous accesses. Moreover, these systems support parallelism at the level of directories only, which is not as beneficial to applications as parallelism at the level of files.

A recent approach to file system design has been to decouple data and metadata [17][39][47][10]. In these systems, metadata and data are served by different sets of servers. *Google File System* (GFS) [17] is a DFS for data intensive applications, custom-built for the application workload and technical environment at Google. A single master maintains all file metadata and several chunkservers store file contents. *Lustre File System* [39] and *Panasas File System* [47] are similar systems that decouple data and control and use Object based Storage Devices (OSD) for their storage needs. Requiring specialized object storage nodes precludes the usage of the storage capacities of most edge nodes. *Parallel Virtual File System* (PVFS) [10] is an open source DFS that attempts to provide high performance and scalable file system services for node clusters. While data is spread out among a large number of cluster nodes, metadata is still served by a single server in PVFS1, and by a small set of servers in PVFS2. All these systems maintain a single, or a fixed set of metadata servers. This imposes a constraint on the scalability of the system and also affects performance. For several applications, performance depends on the speed with which file and directory metadata can be accessed and updated.

Farsite [3] is another DFS which decouples data and metadata. File system namespace is maintained using a collection of replica groups arranged in the form of a tree. File metadata is replicated and stored on the directory group nodes in a Byzantine fault tolerant manner. As with other hierarchy traversal systems, locating the directory group for a file deep in the hierarchy may require several hops, thus making metadata access expensive. *Ceph* [44] is an OSD based DFS which uses a pseudo random function called CRUSH [45] for the distribution of data among nodes in the object storage cluster. The usage of CRUSH rules out the possibility of considering specific node characteristics while making data placement decisions.

OceanStore [24] is a global scale data storage utility that uses untrusted infra-structure. Data objects are stored on primary and secondary tier nodes. Updates are serialized among the primary tier nodes using a Byzantine fault tolerant algorithm and propagated to the secondary tier nodes using a dissemination tree. High churn rates among the primary tier nodes can result in expensive maintenance overheads. The overheads involved in the maintenance of two tiers of nodes and a tree for each data object can also be significantly high. Moreover, every update results in the creation of a new version which is archived in the system, making the system inefficient for large sized files. *Ivy* [29] is a P2P read/write file system based on logs. Each participant maintains a log with information about all the changes made to the data and metadata in the file system. A participant appends changes only to its own log. However, it reads from the logs of other participants and maintains a total order by using version vectors. As a result, Ivy is suitable only for a small number of participants.

JuxMem [5] is a hierarchical platform that supports mutable data sharing services for Grid computing. The hierarchy includes a set of peer groups (clusters) and an overlay network consisting of cluster managers from the peer groups. A data group is created for each data block that is stored in the system. The group handles tasks such as replication, consistency maintenance, data access, etc. The overheads of maintaining individual groups for the data items can affect the system's scalability with respect to the number of data items. Moreover, JuxMem largely ignores namespace management issues and the task of optimizing metadata access.

7. Conclusions and Future Work. In this paper, we have discussed the design of a distributed file system built on top of Internet edge nodes. The DFS attempts to address the needs of widely spread scientific communities that share and run high performance computing applications on large data files. The system allows data placement decisions to be based on user's preference of storage resource characteristics and locations. The two layered architecture allows for the efficient discovery and usage of system-wide storage resources with specific characteristics. ENFS supports flexible consistency management and access semantics to improve application performance. We demonstrate a mechanism that allows a user to continue performing computations on a large dataset, while it is being modified by another user. The limitation, however, is that it can be applied only to certain types of application.

ENFS needs to be further tested by obtaining performance measurements from a larger and wider network of nodes, resembling a real-world scenario. The design of ENFS can be exploited for efficient file discovery. Files can be associated with tags or keywords that describe their contents. The tags to clusters mapping can be stored in the structured overlay in much the same way as it is done for the discovery of far-off resources with specific characteristics. This significantly reduces the number of clusters that must be queried for discovering relevant

files. A group of HPC enthusiasts are working towards defining a few POSIX file system APIs conducive to high performance computing applications [1]. A possible future work is to make ENFS support the proposed POSIX extensions.

REFERENCES

[1] *POSIX extensions*. `http://www.pdl.cmu.edu/posix`, 2006.

[2] *Biomedical Informatics Research Network*. `http://www.nbirn.net`, 2007.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, *Farsite: Federated, available, and reliable storage for an incompletely trusted environment*, SIGOPS OSR, 36 (2002), pp. 1–14.

[4] S. R. Alam and J. S. Vetter, *An analysis of system balance requirements for scientific applications*, in Proc. of ICPP '06, 2006, pp. 229–236.

[5] G. Antoniu, L. Boug, and M. Jan, *Juxmem: An adaptive supportive platform for data sharing on the grid*, Scalable Computing: Practice and Experience, 6 (2005), pp. 45–55.

[6] Y. Bejerano, Y. Breitbart, M. Garofalakis, and R. Rastogi, *Physical topology discovery for large multisubnet networks*, in Proc. IEEE INFOCOM 03, 2003, pp. 342–352.

[7] V. Bharadwaj, T. G. Robertazzi, and D. Ghose, *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society, 1996.

[8] D. Borthakur, *The Hadoop Distributed File System: Architecture and design*. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`, 2008.

[9] J.-M. Busca, F. Picconi, and P. Sens, *Pastis: A highly-scalable multi-user peer-to-peer file system*, in Euro-Par, 2005, pp. 1173–1182.

[10] P. H. Carns, Walter, R. B. Ross, and R. Thakur, *PVFS: A parallel file system for linux clusters*, in Proc. 4th Annual Linux Showcase and Conference, 2000, pp. 317–327.

[11] A. Chien, B. Calder, S. Elbert, and K. Bhatia, *Entropia: Architecture and performance of an enterprise desktop grid system*, JPDC, 63 (2003), pp. 597–610.

[12] B. Cochran, *Distributed file systems: History, taxonomy, and cutting edge ideas*, tech. report, University of Illinois at Urbana-Champaign, 2004.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, *Vivaldi: A decentralized network coordinate system*, in Proc. of the ACM SIGCOMM'04 Conference, 2004, pp. 15–26.

[14] J. Dean and S. Ghemawat, *MapReduce: simplified data processing on large clusters*, Commun. ACM, 51 (2008), pp. 107–113.

[15] D. Eastlake and P. Jones, *US Secure Hash Algorithm 1 (SHA1)*. RFC Editor, USA, 2001.

[16] M. R. Fahey and J. Candy, *GYRO: A 5-D Gyrokinetic-Maxwell Solver*, in Proc. of Supercomputing '04, 2004, p. 26.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*, SIGOPS OSR, 37 (2003), pp. 29–43.

[18] R. Govindan and H. Tangmunarunkit, *Heuristics for Internet map discovery*, in Proc. IEEE INFOCOM 00, 2000, pp. 1371–1380.

[19] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, *Data management in an international data grid project*, in Proc. of First IEEE/ACM International Workshop on Grid Computing, Springer-Verlag, 2000, pp. 77–90.

[20] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West, *Scale and performance in a distributed file system*, SIGOPS OSR, 21 (1987), pp. 1–2.

[21] J. H. Howard, *An overview of the Andrew File System*, in USENIX Winter, 1988, pp. 23–26.

[22] D. J. Kerbyson and P. W. Jones, *A performance model of the Parallel Ocean Program*, Int. J. High Perform. Comput. Appl., 19 (2005), pp. 261–276.

[23] T. Kosar and M. Livny, *Stork: Making data placement a first class citizen in the grid*, in Proc. of ICDCS '04, 2004, pp. 342–349.

[24] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, *Oceanstore: An architecture for global-scale persistent storage*, SIGARCH Comput. Archit. News, 28 (2000), pp. 190–201.

[25] L. Lamport, *The part-time parliament*, Trans. Comput. Syst, 16 (1998), pp. 133–169.

[26] P. J. Leach and D. C. Naik, *A Common Internet File System Protocol (CIFS)*. Internet draft, Internet Engineering Task Force (IETF), 1997.

[27] E. Lua, J. Crowcroft, and M. Pias, *Highways: Proximity clustering for scalable peer-to-peer networks*, in Proc. 4th Int. Conference on P2P Computing, 2004, pp. 266–267.

[28] Y. Luo and K. W. Cameron, *Instruction-level characterization of scientific computing applications using hardware performance counters*, in WWC '98: Proc. of the Workload Characterization: Methodology and Case Studies, IEEE Computer Society, 1998, p. 125.

[29] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, *Ivy: A read/write peer-to-peer file system*, SIGOPS Oper. Syst. Rev., 36 (2002), pp. 31–44.

[30] B. K. Pasquale and G. C. Polyzos, *Dynamic I/O characterization of I/O intensive scientific applications*, in Proc. of Supercomputing '94, 1994, pp. 660–669.

[31] K. Ponnavaikko and D. Janakiram, *Overlay network management for scheduling tasks on the grid*, in ICDCIT, 2007, pp. 166–171.

[32] K. Ranganathan and I. Foster, *Simulation studies of computation and data scheduling algorithms for data grids*, Journal of Grid Computing, 1 (2003), pp. 53–62.

[33] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, *Load balancing in structured P2P systems*, in Proc. of IPTPS 2003, 2003, pp. 68–79.
[34] M. V. Reddy, A. V. Srinivas, T. Gopinath, and D. Janakiram, *Vishwa: A reconfigurable P2P middleware for grid computations*, in Proc. of ICPP '06, 2006, pp. 381–390.
[35] R. Rivest, *The MD5 Message-Digest Algorithm*. RFC Editor, USA, 1992.
[36] D. Roselli, J. R. Lorch, and T. E. Anderson, *A comparison of file system workloads*, in Proc. of USENIX Annual Technical Conference, 2000, pp. 4–4.
[37] A. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, in Middleware '01, Nov. 2001, pp. 329–350.
[38] 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., 1985, pp. 119–130.
[39] P. Schwan, *Lustre: Building a file system for 1000-node clusters*, in Proc. of Linux Symposium, 2003, pp. 380–386.
[40] A. V. Srinivas and D. Janakiram, *Scaling a shared object space to the Internet: Case study of Virat*, Journal of Object Technology, 5 (2006), pp. 75–95.
[41] I. Stoica, R. Morris, D. Karger, F. M. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for Internet applications*, in Proc. of ACM SIGCOMM '01, vol. 31, October 2001, pp. 149–160.
[42] J. Stribling, E. Sit, M. F. Kaashoek, J. Li, and R. Morris, *Don't give up on distributed file systems*, in Proc. of the 6th IPTPS, Feb 2007.
[43] A. Takefusa, O. Tatebe, S. Matsuoka, and Y. Morita, *Performance analysis of scheduling and replication algorithms on grid datafarm architecture for high-energy physics applications*, in Proc. of 12th IEEE HPDC, 2003, pp. 34–43.
[44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, *Ceph: A scalable, high-performance distributed file system*, in OSDI, 2006, pp. 307–320.
[45] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, *CRUSH: Controlled, scalable, decentralized placement of replicated data*, in Proc. of Supercomputing '06, 2006, pp. 31–31.
[46] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, *Dynamic metadata management for petabyte-scale file systems*, in Proc. of Supercomputing '04, 2004, pp. 4–4.
[47] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, *Scalable performance of the Panasas parallel file system*, in Proc. of the 6th USENIX Conference on File and Storage Technologies, 2008, pp. 1–17.
[48] Q. Xu and J. Subhlok, *Automatic clustering of grid nodes*, in Proc. of GRID '05, IEEE Computer Society, 2005, pp. 227–233.

Edited by: Thomas Ludwig
Received: June 16, 2008
Accepted: March 9, 2009