J Grid Computing (2006) 4: 323–341
DOI 10.1007/s10723-006-9035-7

Kosha: A Peer-to-Peer Enhancement for the Network File System*

Ali R. Butt · Troy A. Johnson · Yili Zheng · Y. Charlie Hu

Received: 1 September 2005 / Accepted: 24 February 2006 / Published online: 1 June 2006

Abstract The storage needs of modern scientific applications are growing exponentially, and designing economical storage solutions for such applications – especially in Grid environments – is an important research topic. This work presents Kosha, a system that aims to harvest redundant storage space on cluster nodes and user desktops to provide a reliable, shared file system that acts as a large distributed storage. Kosha utilizes peer-to-peer (p2p) mechanisms to enhance the widely-used Network File System (NFS). P2P storage systems provide location transparency, mobility transparency, load balancing, and file replication – features that are not available in NFS. On the other hand, NFS provides hierarchical file organization, directory listings, and file permissions, which are missing from p2p storage systems. By blending the strengths of NFS and p2p storage systems, Kosha provides a low overhead storage solution. Our experiments show that compared to unmodified NFS, Kosha introduces a 3.3% fixed overhead and 4.5% additional overhead as nodes are increased from two to sixteen. For larger number of nodes, the additional overhead increases slowly. Kosha achieves load balancing in distributed directories, and guarantees 99.99% or better file availability.

Key words distributed storage · fault tolerance · load balancing · location transparency · mobility transparency · NFS · peer-to-peer

1. Introduction

The computational Grid [17], popularized by systems such as Globus [16] and Condor [21], provides ways for applications to be spread across multiple administrative domains, with the goal of catering to the exponentially growing computational and storage needs of modern scientific applications. Typically these systems employ off-the-shelf equipment to provide an economically viable solution. In this context, peer-to-peer(p2p) approaches [6] have also been used to enable dynamic computational resource sharing. Building on similar principles, we propose an economical and fault resilient storage solution for the ever increasing storage demands of applications.

The design of our storage solution is aimed at providing storage for individual Grid sites consisting of multiple nodes, e.g., clusters connected to

*This work was supported in part by an NSF CAREER award (ACI-0238379).
Troy A. Johnson was supported by a U.S. Department of Education GAANN doctoral fellowship.

A. R. Butt · T. A. Johnson · Y. Zheng · Y. C. Hu (✉)
School of Electrical and Computer Engineering,
Purdue University,
West Lafayette, IN, USA
e-mail: ychu@purdue.edu

the Grid. The solution provides an economical and fault-tolerant alternative to the dedicated storage within a single administrative domain. It is not intended to run across domains over the wide-area network. For supporting data movement across such domains, standard Grid data movement protocols can be used. The proposed distributed storage solution can be used in the following scenarios:

- To store applications and data for Grid sites that want to utilize computation resources within the same administrative domain, e.g., Condor job submission nodes;
- To store application generated output data for nodes that execute the applications, e.g., Condor job execution nodes; and
- To store generic user data, e.g., user home directories, for nodes within an administrative domain.

The targeted academic and corporate Grid sites typically utilize off-the-shelf equipment, e.g., desktop machines and rack-mount cluster nodes, primarily for fulfilling computing needs of users, and use centralized servers such as the Network File System (NFS) [7, 29], to cater to the storage needs. However, typical modern compute nodes have a large amount of free disk space, which is wasted in these setups. Hence, our first design goal is *to utilize the cheap storage that is available in such environments to create a distributed file system*.

Because the distributed file system stores files on the disks of peer nodes which may fail over time and may have different disk capacity, our second design goal is *to provide features of location transparency, mobility transparency, load balancing, and high availability through file replication and transparent fault handling*.

The targeted environments also have extensive NFS cross-mounting facilities established to provide users access to storage beyond their local disks. Therefore, it is not practical to replace NFS. (For similar reasons, switching to AFS [19] or xFS [3] may not be possible.) As the widespread use of NFS is indispensable, our third design goal is *to retain the widely used NFS semantics, so that users and applications can access the distributed file system without any changes*.

In this paper, we propose Kosha, a distributed file system that employs peer-to-peer (p2p) technology and the disk space available on participating nodes to enhance NFS. Kosha provides a single file system image identical to NFS, requires only minimal configuration, and does not entail changes to the underlying operating system. The result is a simple yet effective system, which is readily deployable, does not burden the user with the need to learn a new interface, and supports unmodified applications.

Kosha organizes the participating nodes into a structured p2p overlay, and uses NFS facilities to make the files available across peers. It ensures that the location of the files remains transparent to the user. Unique to the design of Kosha is that instead of distributing individual files over the distributed storage provided by the nodes in the p2p overlay, it distributes at the level of directories, i.e., files in the same directory are by default stored in the same node as that directory. Furthermore, Kosha controls the granularity of directory distribution via a parameter that controls the depth beyond which subdirectories are not distributed, i.e., they are stored on the same node as their parent directory. As we will show, distribution at the directory level allows Kosha to impose less overhead on NFS operations, while achieving load balancing comparable to distribution at the file level.

Kosha and standard NFS can safely co-exist on a node and their operations do not interfere with each other. It is up to the node owner to decide the portion of a node's storage space that is used for Kosha and standard NFS. In order for users to reap the benefits of Kosha, they should explicitly store their files under the Kosha mount points. This process can be simplified if the system administrators move all the users' files to Kosha mount points and set their home directories to those on Kosha. In this way, users can transparently benefit from Kosha features.

The main contributions of this work are as follows:

1. The aggregation of unused disk space on many computers into a single, shared file system with standard NFS semantics;
2. Load balancing via an efficient scheme of distributing directories instead of files;

3. High availability through replication and transparent fault handling; and
4. A detailed evaluation of the approach, including its performance compared to unmodified NFS, and its ability to provide load balancing and fault tolerance.

The rest of the paper is organized as follows. Section 2 presents the enabling technologies for Kosha. Section 3 presents the main idea of file distribution across multiple nodes in Kosha. Section 4 presents the design of Kosha and how it handles various NFS operation. Section 5 describes our prototype implementation. Section 6 presents a detailed evaluation of Kosha. Section 7 discusses the related work. Finally, Section 8 gives concluding remarks.

2. Enabling Technologies

Two aspects, advancement in hardware technology and p2p routing algorithms, serve as enabling technologies for our proposed approach. In the following sections we discuss these aspects in more detail.

2.1. Large Unused Local Disk Space on Desktops

Most desktop computers in today's academic or corporate environments are purchased mainly for processing power. Nevertheless, standard packages, which are rampant in such environments, usually ship with large capacity disk drives [12, 14]. In order to support our conjecture that a large amount of disk space is wasted in the focused environments, we performed a survey of over 500 instructional machines at Purdue University. The survey showed that more than 80% of machines have 1.5 GHz Intel Pentium 4 or better processors, and the total available disk space ranged from 8 GB (for older systems) to 60 GB (for the latest systems). A little over 84% of the machines have a local disk of 40 GB; however, the local disk utilization is only up to 4 GB for holding the operating system and temporary user files. For the systems that have at least 40 GB disk space, at least 90% of the local disk space on each machine is unused. As disks become cheaper and larger in capacity, this wastage is bound to worsen. On the other hand, the three NFS servers used by these machines have about 75% space being used. The servers have to impose strict quotas in order to avoid being full. Such central servers require regular addition of new disk space to accommodate new users and the ever growing storage needs of many users, an obviously expensive and cumbersome procedure.

Simply running NFS servers on every machine that has unused local disk space and letting users share the space is far from practical. The maintenance of a huge number of servers can be inhibiting, and human interaction and configuration errors may poorly affect the performance. Furthermore, if all nodes are NFS servers, users must remember on which machines their files are stored – a difficult and cumbersome task if many machines are used for storage. Symbolic links can help the user to locate their files quickly, but stale links can make the situation even more confusing. Another issue is that NFS does not provide redundancy, so if machines fail or are taken offline for maintenance, the information stored on them becomes inaccessible. The failure often causes other machines (repeatedly trying to access the failed machines) to respond slowly to requests they receive – an effect which spreads rapidly to degrade the performance of the entire system. The users may retrieve files from a daily or weekly backup storage, but in a large organization it may not be economically feasible to backup the data from the local disks of all machines. These observations stress the opportunity of, and the need for, a utility layer above NFS to manage locally available disk space as an economical way of fulfilling the ever growing storage demands of users.

2.2. Structured P2P Overlay Networks

Structured p2p overlay networks such as CAN [24], Chord [32], Pastry [26], and Tapestry [34] effectively implement scalable and fault tolerant *distributed hash tables* (DHTs), where each node in the network has a unique node identifier (`nodeId`) and each data item stored in the network has a unique key. The `nodeId`*s* and keys live in the same name space, and each key is mapped to a unique node in the network. Thus DHTs

allow data to be inserted without knowing where it will be stored and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored.

The key aspects of these structured p2p overlays are self-organization, decentralization, redundancy, and routing efficiency. Self-organization promises to eliminate much of the cost, difficulty, and time required to deploy, configure and maintain large-scale distributed systems. The process of securely integrating a node into an existing system, maintaining its integrity invariants as nodes fail and recover, and scaling the number of nodes over many orders of magnitude is fully automated. The heavy reliance on randomization (from hashing) in the `nodeId` and key generation provides good load balancing, diversity, redundancy and robustness without requiring any global coordination or centralized components, which could compromise scalability. In an overlay with N nodes, messages can be routed with $O(\log N)$ overlay hops and each node maintains only $O(\log N)$ neighbors.

The functionalities provided by DHTs allow for transparent distribution of files on multiple servers. In the next section, we discuss how this facility is used in the Kosha design. While any of the structured DHTs can be used to implement file distribution in Kosha, we use Pastry for this paper. In the following, we briefly explain the DHT mapping in Pastry.

Pastry. Pastry [8, 26] is a scalable, fault resilient and self-organizing p2p substrate. Each Pastry node has a unique, uniform, randomly assigned `nodeId` in a circular 128-bit identifier space. Given a message and an associated 128-bit key, Pastry reliably routes the message to the live node whose `nodeId` is numerically closest to the key.

In Pastry, each node maintains a routing table that consists of rows of other nodes' `nodeId`*s* which share increasingly longer prefixes with the current node's `nodeId`. In addition, each node maintains a leaf set, which consists of l nodes with `nodeId`*s* that are numerically closest to the present node's `nodeId`, with $l/2$ larger and $l/2$ smaller `nodeId`*s* than the current node's `nodeId`. The leaf set ensures reliable message delivery and is used to store replicas of application objects. Pastry routing is prefix-based. At each routing step, a node seeks to forward the message to a node whose `nodeId` shares with the key a prefix that is at least one digit longer than the current node's shared prefix. The leaf set helps to determine the numerically closest node once the message has reached the vicinity of that node. A more detailed description of Pastry can be found in [8, 26].

3. Distribution of a File System Across Nodes

Kosha distributes data to various nodes that participate in storage-space sharing by joining the Pastry overlay. The virtual mount point */kosha* serves as an access point to the distributed file system provided by Kosha. On each node, the directory */kosha_store* serves as the storage for Kosha. From a user's perspective, the */kosha/$USER* directory actually corresponds to the union of the */kosha_store/$USER* directories on all nodes, as shown in Figure 1. For Kosha *$USER* is the same as in NFS, i.e., the user's home directory and is typically the same as the login of the user.

3.1. Directory Distribution Across Multiple Nodes

To achieve load balancing, Kosha employs hashing provided by Pastry and distributes directories created under */kosha* to multiple nodes. It is assumed that all the files in a directory reside on the same node, i.e., the node to which the directory name is mapped, except for subdirectories, which reside on the nodes selected via mapping of the subdirectory names. This design helps to reduce

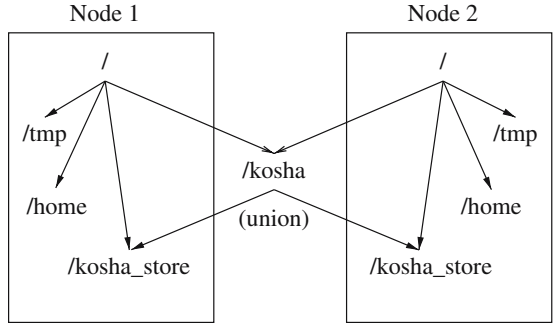

Figure 1 Virtual directory hierarchy: */kosha* is the virtual directory and is the union of */kosha_store* on all the nodes.

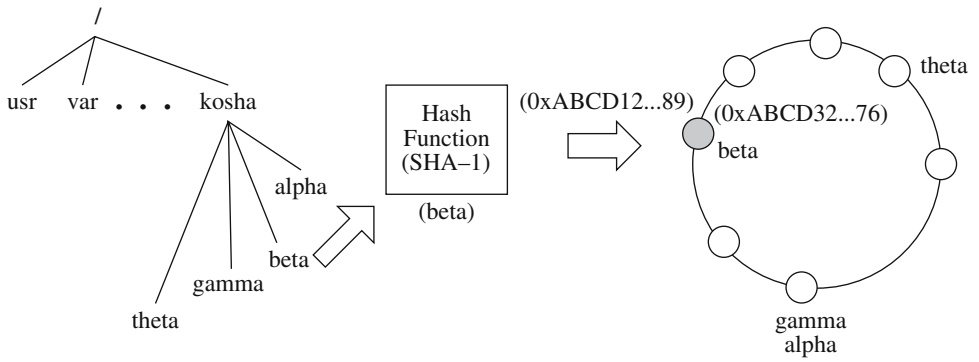

Figure 2 Example of file distribution to multiple nodes. The virtual mount point is */kosha*. The directory name is first hashed using a general hashing function such as SHA-1 to generate a unique key, which is then routed using Pastry to a node whose `nodeId` is numerically closest to the key. The selected Pastry node will provide the physical storage for the directory. The actual file operations, however, are performed via the NFS protocol (not shown).

costs of hashing and subsequent lookups for the actual storage nodes, while maintaining a good load balance.

For example, to locate a node for a file */dir1/file1*, Kosha performs the following mapping:

$$/kosha/dir1/file1$$
$$\Longrightarrow DHT(hash(dir1)) : /kosha\_store/dir1/file1$$

The following steps are done for this purpose. A 128-bit unique key is created via a SHA-1 [1] hash of the directory name. Next, this key is used to lookup a node according to the DHT implementation of the p2p substrate. For example, in the case of Pastry [26], the selected node is the one whose identifier is numerically closest to the key value. The event of key collisions due to two or more subdirectories sharing the same name only implies that the colliding directories will be stored on the same node, and does not pose a problem in distinguishing them, as their paths are unique.

Figure 2 shows an example distribution of directories to various nodes. When a directory is distributed to a node other than the one that stores its parent directory, a soft link to the distributed subdirectory is placed in the parent directory to serve as a place holder for properly listing contents of the parent directory.

3.2. Controlling the Granularity of Distribution

Kosha maintains a system-wide parameter, the distribution level, which dictates how many levels of subdirectories will be distributed to multiple nodes. For instance, distribution level 1 implies that hashing is performed for only direct subdirectories (first level subdirectories) of the virtual file system mount point (*/kosha*) to distribute them to multiple nodes. As a result, all lower level subdirectories are stored on the same node as the node on which their parent directory is stored. For example, with distribution level 1, the directories */kosha/dir1* and */kosha/dir2* may be stored on different nodes as determined by the p2p substrate, but the directories */kosha/dir1/dirX* and */kosha/dir1/dirY* are both stored on the same node as the one which stores */kosha/dir1*. Similarly, distribution level 2 implies that another level of subdirectories will also be distributed to multiple nodes. In this case, */kosha/dir1/dirX* and */kosha/dir1/dirY* may be stored on different nodes than the one that stores */kosha/dir1*. Hence, distribution level controls the granularity of distribution and load balancing.

3.3. Optimization

The heterogeneity in the storage space contributed by nodes and variations in directory size implies that a node selected for storing a directory may not have enough local disk space to hold the directory and all of the files (used here generally to mean subdirectories as well) stored under it. However, it is also not possible to decide the size of a directory a priori, i.e., without knowing the size of

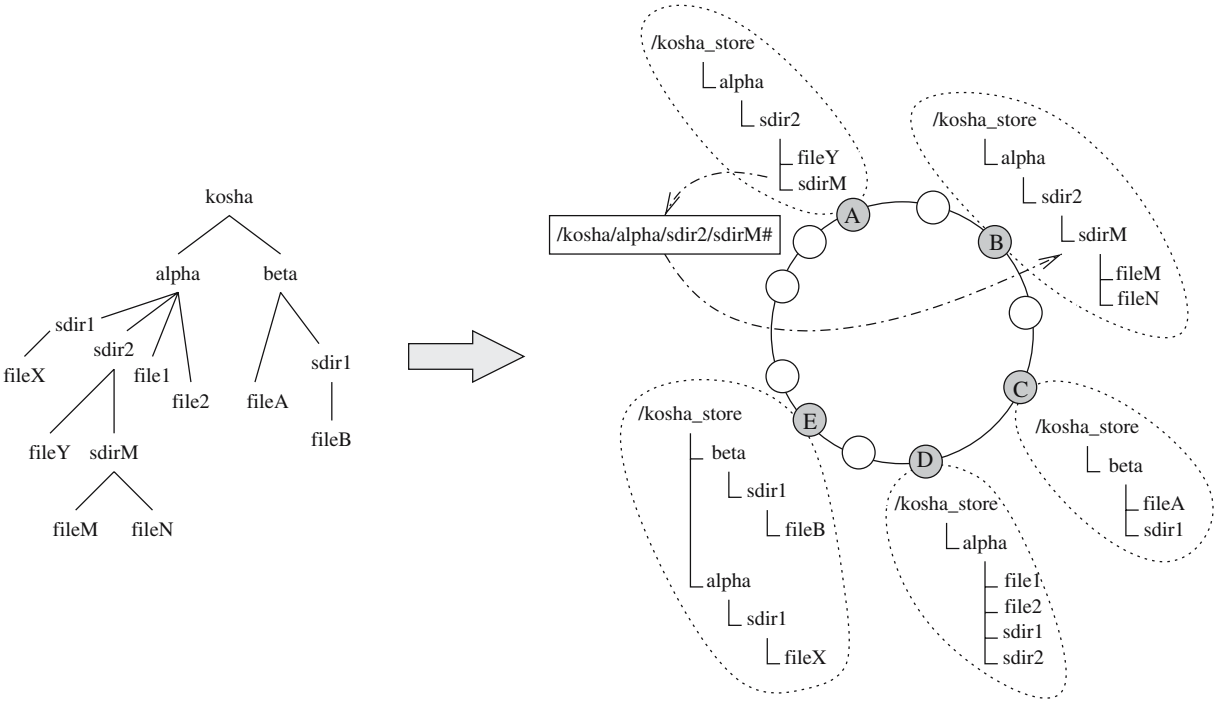

Figure 3 Example of subdirectory distribution to multiple nodes. The files in the same directory are stored on the same node as the parent directory. However, the subdirectories are distributed to remote nodes. The distribution level is set to 2, and *A* has limited capacity which causes */alpha/sdir2/sdirM* to be redirected to *B*. The example contents of the special link are shown in the *rectangle*. Node *B* is chosen for storing the redirected subdirectory because *DHT(hash(sdirM#)) = B*.

the files that will be created in it. To address this, Kosha employs redirection for all newly created files when the local disk space has exceeded a pre-specified utilization level. When this happens, the file is *redirected* and stored on a different node. Redirection is done by concatenating a random salt to the file name, and rehashing the new name to find a suitable node. The redirection due to storage capacity is an iterative operation rather than recursive, i.e., the redirection process repeats till a node with enough disk space is found, or a pre-specified number of retries is exhausted. This approach is derived from a similar approach in PAST [27].

When a file is redirected, a special soft link to the redirected file is created in the parent directory. The special soft link serves to connect the redirected file to its actual location in the parent directory. The name of the link is the same as the name of the redirected file; this helps Kosha list the directory contents of the parent directory. The target of the link is the file name concatenated with the salt value. When Kosha comes across a special link, it follows the special link and accesses the redirected file. This provides users with transparency to redirection. Figure 3 shows an example of file distribution with this optimization.

4. Kosha Design

In Kosha, nodes contributing disk space join a p2p overlay network, and are identified by unique `nodeId`*s* assigned to them via the Pastry interface [26]. The nodes are assumed to run NFS servers, so that their contributed disk space can be accessed via NFS. It is assumed that only the system administrator has full access to these nodes, and the users cannot modify the system arbitrarily.

Various file operations performed on */kosha* are handled as follows. First, Kosha determines the node on which a file is stored by performing the

mapping described in Section 3.1, and following any redirection as necessary. Second, the NFS Remote Procedure Call (RPC) for performing the file operation is modified to occur on */kosha_store* on the selected node, instead of */kosha* on the client node. Kosha does this by forwarding the modified RPC to the selected node. Third, the receiving node performs the operation and returns the results to Kosha, which then records the information needed for future accesses. Finally, Kosha returns control to the client. Hence, the client remains unaware of the underlying RPC forwarding, and the whole operation is transparent, except for a delay caused by the lookup for the appropriate storage node.

4.1. NFS Operations Support

In the following, we describe the semantics supported by Kosha, followed by a discussion of how Kosha handles various NFS operations.

4.1.1. *Semantics in Absence of Failures*

The semantics of Kosha are the same as NFS in the absence of failures. All accesses to a file are guaranteed (by the DHT-based storage node location) to be sent to the same storage node, and therefore, every user sees the same instance of a file. In case of failures, Kosha differs from NFS in that it continues to provide access to files, whereas NFS does not. See Section 4.3 for more on the failure semantics of Kosha. The behavior of Kosha in the presence of client caching also remains the same as that of NFS.

4.1.2. *Virtual File Handles*

NFS uses file handles to access files. These handles are opaque, i.e., they only have meaning to the NFS server, implying that the clients can be given any identifier for a file as long as it corresponds uniquely to a file handle in the server. The opacity provides Kosha with a way to decouple actual file handles from identifiers handed to the clients. We refer to these identifiers as *virtual file handles*, as they serve to access files in the */kosha* virtual file system. Kosha maintains a table of mappings from virtual file handles to real file handles, which allows it to provide location transparency. As explained below, Kosha also stores the full file path for each entry in the table. *The extra level of indirection enabled by the use of virtual handles allows Kosha to transparently substitute handles for file replicas in the event of node failures.*

4.1.3. *Locating Files*

In NFS, a `lookup` RPC is used to obtain a file handle for a file. The RPC contains the handle for the parent directory, and the name of the file for which the lookup is desired. Note that in NFSv3, the RPC does *not* contain the full path to the file. Once a handle is available, other NFS operations can be performed on the file or directory by presenting the handle. Looking up the full path by an NFS client requires a sequence of `lookup` RPCs, unless the handles for the ancestor directories have already been cached.

To perform a `lookup` RPC from the local NFS client, Kosha first looks up the full path to the parent directory in the virtual handle table, which is already known because of previous `lookup` calls, and appends the name of the file to the parent directory's full path. Kosha then examines the full path to the file and uses the distribution level to determine what directory name should be used for node lookup. Performing this lookup gives Kosha the remote node R on which the file is stored.

Next, Kosha looks up the entire path on R, as if it is an NFS client of R. Finally, when the call returns with a handle, a virtual handle (as described in Section 4.1.2) is created, and the virtual handle is given to the client.

All subsequent RPCs that supply the virtual handle are mapped to the real handle to perform the NFS operation. For instance, RPCs such as `read`, `write`, `getattr`, and `setattr` provide the virtual handle, which is mapped by Kosha to the actual handle, and the operations can then be completed.

4.1.4. *File Creation and Renaming*

To create files or directories, the first step is to locate the node on which the newly created files should be stored. To create a file, Kosha locates the node R to which the parent directory

is mapped and the handle to the parent directory on the node as in Section 4.1.3, and then sends a message with the client provided RPC parameters to *R*. *R* uses this information to create the file, which becomes the primary replica of the file, and returns the file handle of the created file to Kosha. Kosha stores the returned handle in the virtual handle table, and returns the corresponding virtual handle to the client, completing the RPC.

The creation of a subdirectory that is below the distribution level is similar to the file creation process described above. If the subdirectory is within the distribution level and thus needs to be distributed, the remote node *R* is first located by hashing the directory name using the DHT. One or more RPCs are then sent to *R* to create the new subdirectory as well as all the missing ancestor directories in the hierarchy on *R*.

A `rename` operation on a file does not imply migration to a different node as all files in a subdirectory reside on the same node. Therefore, if the file is not redirected, `rename` is performed as in the standard NFS. If the file is redirected, a special link is present, e.g., in Figure 3, *A:/kosha_store/alpha/sdir2/sdirM* (assume it is a file) points to the file on *B*. In this case, the `rename` is achieved by renaming the link and the actual file, e.g., */kosha_store/alpha/sdir2/sdirM* to */kosha_store/alpha/sdir2/sdirM_NewName* on both *A* and *B*. The target of the link needs not be changed, because $DHT(hash(sdirM\#)) = B$ remains true. This prevents unnecessary moving of files on each `rename` call, and yields an efficient solution. The same process is used for renaming subdirectories that are not distributed.

Renaming of distributed subdirectories is complex, and in essence is equivalent to a *copy* to a new location followed by a *delete* of the old location. The process involves traversal of all subdirectory levels on all replicas and is expensive.

4.1.5. *Removing Files*

An NFS client uses `remove` or `rmdir` RPCs to delete files or directories, respectively. To delete a file, the first step is to determine the remote node on which the file is stored by looking up the virtual handle mapping. Next, Kosha forwards the RPC to the remote node, where it is processed and the file is removed. Once again, as in the previous Kosha operations, the reply values are returned to Kosha and finally to the client.

The directories that are not distributed are deleted in a similar manner. To delete a distributed subdirectory, Kosha first deletes the subdirectory from the node on which it is stored. It then examines the empty directory structure created to support the distributed subdirectory for possible use by other subdirectories with some common path prefix. Any portion of the empty hierarchy that is not shared with other subdirectories is then removed. Finally, the soft link to the deleted subdirectory in the parent directory is deleted to ensure proper listing of parent directory contents. This action completes the directory deletion process.

4.1.6. *Security*

Security in Kosha is identical to NFS since files in Kosha maintain their permissions. Also, in most of the targeted academic or cooperate networks, the users either are not given administrative access to their machines, or NFS servers are not run on such machines. Therefore, it is safe to assume that the files stored on distributed nodes are at least as secure as on a central NFS server. For added security, however, Kosha can be extended to support a majority consensus system based on Byzantine agreements [10], as utilized in [28]. The performance of the system may be sacrificed, if the need for supporting mutually untrusted nodes arises. The p2p substrate can support tighter security extensions [9]; however, in our implementation we did not incorporate such approaches.

4.2. Managing Replicas

Kosha maintains *K* replicas of a file on the neighboring *K* nodes in the node identifier space. The random assignment of node identifiers ensures that the replicas are dispersed fairly and can provide good fault tolerance. Neighbors in the node identifier space have no relationship in terms of physical proximity.

For each file, there is a node that is located using the techniques of Section 3; we refer to this node as the *primary replica*. All accesses to the

file are sent to the primary replica. The primary replica is responsible for maintaining K replicas of the file on K neighboring nodes in its leaf set. The replicas are inaccessible to the local users, as they may accidentally or maliciously modify the replica. The directory hierarchy structure (containing the file) is replicated along with the file on the replica nodes. In addition, special links in the same directory, i.e., those pointing to distributed subdirectories, are replicated as well.

The primary replica is also responsible for removing files from all replicas when they are deleted. When the primary replica receives an RPC for deleting a file, it removes the file locally and also forwards the RPC to all the replicas, hence removing all instances of the file. If a replica node fails before performing the delete operation, it does not create any inconsistency as explained in the node failure discussion below.

It should be noted that with the present design the primary replica is in charge of all file operations unless it fails and a new primary replica is selected. Since there are K replicas of the files available, there is potential for performance improvement by leveraging these replicas. We currently are exploring optimization techniques that allow at least read operations to be served from any one of the K replicas.

The fault tolerance and replica management may be affected by high degree of node disconnections and joins to the p2p overlay. Since the participating nodes in our targeted environment are desktop machines or cluster nodes running a flavor of UNIX, we also expect that they will behave in a stable manner in the p2p overlay with mean time to failure in the order of days. That is, the expected number of disconnections or failures will be low for each participating node. Hence, the replica management is not expected to become overwhelming and restrict the applicability of Kosha.

4.3. Node Addition and Failure

The p2p component of the system handles nodes joining or leaving (including failure) the system at will, and informs Kosha on a node N when nodes in N's leaf set are affected. Kosha then dynamically adjusts the file distribution to maintain proper locations of the primary replica and the K additional replicas.

4.3.1. *Primary Replica*

Pastry evenly divides the key space between adjacent nodes in the circular identifier space, with the node with `nodeId` numerically closest to the file key responsible for the file. In case of node addition, action is required only at the two nodes that become immediate neighbors of the new node. If N is one of these neighbors, the key space distribution changes for it, implying that some of the files, for which N is the primary node, now belong to its new neighbor and should be moved. Kosha examines the files stored on N and the N's leaf set to determine which files need to be moved. If a move is required, the files are copied to the new node, and their copy on N becomes one of the replicas. The migration of files ensures that a new node always has the files for which it is the primary node.

4.3.2. *Additional Replicas*

In case a replica node fails, or a new node is added, Pastry detects the change and informs the local Kosha of the event via a callback function. In response to this, the local Kosha creates a copy of its contents for which it is the primary replica, and sends the copy to the newly added node, which now serves as one of the K replicas.

Note that since a node can be revived with a different identifier which places it in a different location in the p2p node identifier space, all Kosha data on a revived node is purged. Purging ensures that nodes do not end up accumulating replicas from their previous locations in the p2p identifier space as they fail and recover.

4.4. Transparent Fault Handling

Our implementation of Kosha assumes all failures to be crash failures, and transparently handles the failure of a primary replica node as follows. The client has a virtual handle to the file, which Kosha transparently can change to index the handle of a file replica when the primary replica node fails. The following sequence of events occur in case

of such failure. When any client accesses a file whose primary replica has failed, Kosha detects an RPC error and removes the mapping for the virtual handle. It then proceeds as though a `lookup` RPC was made and locates the handle for another replica of the file. The p2p-based replication of Section 4.2 guarantees that the lookup automatically will be sent to a node that already stores a valid replica. An error occurs when no valid replica for the file can be found. By effective replication Kosha provides very high availability, and due to the highly randomized physical location of the neighbors in the node identifier space, there is a high probability of finding a replica even under a large number of node failures.

Another interesting scenario may occur when the primary replica fails while performing content migration (due to either a node join or failure) to a newly inducted replica node. With the design described so far, the new replica may not have the correct contents. To overcome this problem, when a primary replica performs migration it also creates a file named `MIGRATION_NOT_COMPLETE` on the node to which content is being migrated, and removes it after the migration completes. In the case of failure of the primary replica before migration is completed, the file `MIGRATION_NOT_COMPLETE` serves as a flag to indicate problems with content migration. The new primary replica checks for the existence of this file on all the K replicas, and perform the content migration as before to make all the replicas current. In this way, fault tolerance is achieved even for this scenario.

Finally, storing a mapping from virtual handles to real handles means Kosha is not stateless. But this mapping is only provided as a service to the kernel, and due to our crash failure assumption, if Kosha fails, the entire machine including the kernel must have failed. Therefore, virtual handles need not be persistent.

4.5. Load Redistribution

The DHT-approach adopted in Kosha provides good average load-balance, but it is possible to have extreme scenarios, where some nodes operate near capacity while the available space on neighboring nodes is minimally used. To address the node overload problem, the redirection of Section 3.3 can be used to redirect files to a different node. However, at high system utilization, the number of redirections can become large and results in system performance degradation. To avoid this problem, we utilize a proactive approach to load balancing based on periodic redistribution of files to neighboring nodes.

Once a node is loaded beyond a pre-specified fraction of total available space on it, it attempts to shift some of its stored files to its neighbors as follows. The node communicates with its left and right neighbors in the identifier space, and if a neighbor has available capacity, files are proactively redirected to that neighbor, freeing up space on the node. In the presence of K-replicas of a file, the load redistribution is done with $(K+1)$th neighbor to ensure that the file as well as its replica is moved to other nodes.

It is possible that multiple nodes decide to redirect files to a particular node and cause it to become overloaded. To avoid such an occurrence, each over-loaded node randomly determines a waiting period before attempting to redistribute its files. Moreover, nodes only allow one remote node at a time to transfer files to them. This ensures that redistribution will not cause a normally loaded node to become overloaded.

4.6. Quota Management

In the distributed file system we have described so far, users are at liberty to utilize all of available space without limit. This is not desired as users can maliciously or accidentally consume all the available space and affect other users. To address this problem, we now describe a quota management system that Kosha uses to limit the disk space usage of individual users.In contrast to the local disk quota management approach of standard UNIX that enforces hard limits by monitoring each disk I/O of a user, we adopt a scheme that enforces soft limits but avoids the overhead of monitoring distributed I/Os of Kosha. At the heart of our scheme is a daemon that runs periodically (e.g. every 30 min) and determines the amount of disk space consumed by a user. The quota daemon runs on all participating Kosha nodes. On each

node this daemon determines the disk usage of the user directories that have their root directory */kosha/$USER* stored on the node. To perform this enumeration the daemon can utilize a simple command such as `du` to determine the disk usage of */kosha/$USER* and all of its subdirectories. Note that Kosha provides automatic handling of the enumeration of distributed subdirectories, as it redirects I/O requests on such directories to the proper remote nodes.

Once the daemon determines the disk usage of a user, it stores this information in a special file */kosha/$USER/MYQUOTA*. The file also contains the disk usage limit for the user. Kosha retains ownership of this file and only allows the user read access to it. The advantage of using a file to store this information is two-fold: It eliminates the need for a centralized database for maintaining quota information; and the file is automatically stored and located using already available Kosha mechanisms.

The enforcement of the quota limits is done as follows. Every time a user opens a file for writing, Kosha first consults the */kosha/$USER/MYQUOTA* file to determine whether the disk quota has been exceeded, and if so, refuses any modifications to existing files or creation of new files till the user removes enough files to bring the usage below the quota limit.

In order to change the disk quotas, Kosha administrator can walk the user home directories and update the */kosha/$USER/MYQUOTA* file. Hence, Kosha employs a periodically running daemon to enforce disk quotas. Although the limits are soft – users can continue to exceed their allotted quotas between daemon runs – the scheme provides an effective mechanism to limit unbounded disk space usage.

5. Software Architecture

Each node participating in storage sharing runs an instance of the Kosha software. A local disk partition is created and used for space contribution. The size of the partition provides control over the amount of disk space contributed to Kosha.

The Kosha loopback daemon *koshad* is implemented as two tightly coupled components: An NFS loopback server [22] and one of the p2p routing substrates, Pastry, as shown in Figure 4.

Koshad on each machine is assigned to the same virtual mount point, */kosha*. Afterwards, whenever an application performs a file I/O on any path beginning with */kosha* (step 1 in Figure 4), the NFS portion of the OS kernel will make a remote procedure call (RPC) to the loopback server *koshad* (step 2).

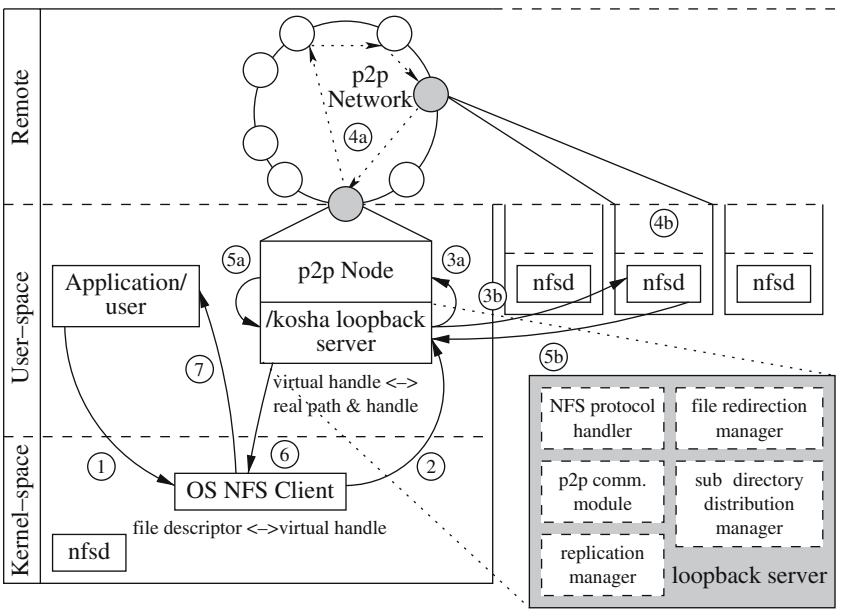

Figure 4 Kosha architecture: **1** application makes an I/O system call, **2** kernel makes an RPC call, **3a** local port request to peer substrate or **b** handle substituted and RPC forwarded, **4a** overlay locates node storing file or **b** file I/O occurs, **5a** local port reply from peer substrate or **b** I/O result returned, **6** RPC returns with virtual handle or result, **7** system call returns control to application.

5.1. Prototype Implementation

The implementation of Kosha is divided into two parts. One part is dedicated to managing p2p communication between nodes and utilizes the Pastry API. The only publicly available version of this API is FreePastry [15] and is written in Java. Therefore, for our experiments we implemented a simplified version of the Pasty API using 800+ lines of C++ code.

The second and larger part called *koshad* handles accesses to the file system and manages NFS RPCs. It is implemented as an NFS loopback server built on top of the SFS toolkit [22] with 4, 000+ lines of C++ code.

In order to start the system, the p2p part is started first, followed by the execution of *koshad*. Once started, *koshad* establishes communication with the local p2p component using sockets. The messaging between the nodes occurs at two levels. The node lookup and other p2p messages are relayed using the p2p substrate. Once a node is chosen for a specific operation, *koshad* uses direct NFS RPCs to communicate with remote NFS servers.

6. Evaluation

In this section, we present experimental results obtained from our prototype implementation of Kosha.

6.1. Performance

To determine the performance of the proposed scheme, we measured Kosha execution times for a modified[1] Andrew benchmark (MAB) [28] and compared it to NFS Version 3. These experiments were performed on a 16-node configuration made up of two sets of machines. The first set contains 4 nodes, each with a 2.0 GHz Intel P4 processor, 512 MB RAM, 40 GB 7, 200 RPM Barracuda Seagate hard disk, and running FreeBSD 4.6. The second set contains 12 nodes, each with a 3.0 GHz Intel Xeon processor, 2 GB RAM, 146 GB 10K RPM U320 SCSI hard disk, and running FreeBSD 4.9. All nodes are connected via a 100 MB/s Ethernet switch. The slower nodes are used only if more than 12 nodes are employed in an experiment. The MAB workload is 51 MB in size, with a maximum subdirectory level of 4.

6.1.1. *Scalability*

Table 1 shows the first set of measurements comparing the performance of Kosha, varying the number of nodes, relative to that of NFS. In this case, the distribution level was fixed at 1, i.e., only the first level directories under */kosha* were distributed to multiple nodes. The level was chosen to remove the effect of subdirectory distribution, and thus isolate the performance overhead due to p2p lookups. The replication factor was also fixed at 1 for similar reasons. Moreover, each node contributed 35 GB of disk space, enough to accommodate all the files to be stored on it, hence eliminating the effect of file redirection. For each overlay size, 10 runs of the benchmark were made, and the average execution time for each phase was recorded. For Kosha, we measured the performance as we successively increased the number of nodes from 1 to 16. The NFS configuration consists of two nodes with one running as a client, and the other running as a server.

The first observation that can be made from the table is that Kosha in a single node setup performs slightly better than NFS for all phases of the benchmark except `compile`, and about 1% better overall. This is because in this configuration all transactions are local to the node and any overhead due to data transfer over the network is avoided. However, the `compile` phase is compute intensive, and running everything (Kosha + compilation) on the same node manifests as an overhead of 4.7% for this phase.

As the number of nodes is increased, the effect of locally stored files is diminished and an overhead is observed. The total overhead introduced by Kosha as number of nodes is increased to 16, as compared to the performance of NFS, is under 6%. Adding more nodes into the system does not affect the overall performance drastically (only 4.5% additional total overhead introduced when

[1] The benchmark was modified to run on FreeBSD with a larger workload.

Table 1 Performance of a modified Andrew benchmark on Kosha with increasing number of nodes.

Benchmark	NFS exec. time	Kosha				
		1 Node	2 Node	4 Nodes	8 Nodes	16 Nodes
		exec. time (overhead)	exec. time (overhead)	exec. time (overhead)	exec. time (overhead)	exec. time (overhead)
mkdir	2.242	2.241 (1.000)	2.243 (1.000)	2.259 (1.008)	2.261 (1.008)	2.304 (1.028)
copy	17.503	16.496 (0.942)	17.401 (0.994)	17.601 (1.006)	17.791 (1.016)	18.207 (1.040)
stat	1.531	1.513 (0.988)	1.533 (1.001)	1.534 (1.002)	1.537 (1.004)	1.595 (1.042)
grep	3.709	3.235 (0.872)	4.026 (1.085)	4.030 (1.087)	4.092 (1.103)	4.181 (1.127)
compile	21.897	22.933 (1.047)	23.231 (1.061)	23.560 (1.076)	23.860 (1.090)	24.254 (1.108)
Total	46.882	46.418 (0.990)	48.434 (1.033)	48.984 (1.045)	49.541 (1.057)	50.541 (1.078)

The table shows average execution times for each phase and the respective overhead of Kosha compared to NFS. The distribution level for Kosha was fixed at 1 for these measurements. All times are in seconds.

the number of nodes increased from 2 to 16), this is because the DHT lookup is always one hop in the small p2p overlay.

6.1.2. *Discussion*

The average overhead D introduced by the design of Kosha can be categorized as:

$$D = I + (H * hc) * \frac{(N-1)}{N}$$

where N is the number of nodes in the network, I is a constant overhead introduced by the interposition code for redirecting NFS calls to different nodes, H is the number of hops a message has to travel to the destination node, and hc is the average latency of each hop. H is a function of N and equals $log_{2^b}(N)$ where 2^b is the base of a digit in Pastry `nodeId` with a typical value of 16 or 32. The factor $\frac{(N-1)}{N}$ (referred to as the *overhead factor* for this discussion) accounts for the percentage of total files stored on remote nodes compared to those stored on the local node. For small N, a higher percentage of files are stored locally and file operations to them are not affected by the network latency. When N is increased initially, the main overhead introduced is from the increase in the number of files served from remote nodes, which becomes constant as N becomes large. For example, when N is increased from 1 to 8, the percentage of remotely stored files increases from 0% to 87.5%, whereas for 16 nodes 93.75% files are stored remotely, an additional increase of only 6.25%. For a typical network of 10,000 nodes, the maximum value of H is 4, hc is under 1ms (this is typical within an organization), and the *overhead factor* ≈ 1. Hence, the overhead D is not expected to exceed 4ms plus a constant factor. This shows that Kosha is highly scalable; additional nodes can be introduced into the system with little increase in the overhead.

6.1.3. *Subdirectory Distribution*

To measure the effect of subdirectory distribution on the overall performance, we varied the distribution level between 1 to 4, while fixing the number of nodes in Kosha to be 8. Once again, 10 runs of the MAB were made, and the average execution times were recorded.

Table 2 shows that the overhead in distribution levels 2, 3, and 4 relative to distribution level 1 are 4.8%, 6.3%, and 6.8%, respectively. This implies that having a large distribution level is not inhibiting. Also observe that the cost on `mkdir` and `copy`

Table 2 Performance of a modified Andrew benchmark on Kosha as the distribution level is increased. For these measurements, the number of nodes was fixed at 8. All times are in seconds.

Benchmark	Dist-level 1	Dist-level 2	Dist-level 3	Dist-level 4
	exec. time	exec. time (overhead)	exec. time (overhead)	exec. time (overhead)
mkdir	2.261	2.483 (1.098)	2.623 (1.160)	2.692 (1.191)
copy	17.791	19.231 (1.081)	19.286 (1.084)	19.301 (1.085)
stat	1.537	1.752 (1.140)	1.786 (1.162)	1.801 (1.172)
grep	4.092	4.297 (1.050)	4.331 (1.058)	4.352 (1.064)
compile	23.860	24.147 (1.012)	24.623 (1.032)	24.770 (1.038)
Total	49.541	51.910 (1.048)	52.649 (1.063)	52.916 (1.068)

is significantly more than on `compile` and `grep`. The reason for this is that when the directories are created in the `mkdir` and `copy` phases, Kosha has to perform two hashes to locate the node on which the subdirectory will be stored, and to locate the parent directory where the special link will be created. Then the empty hierarchy as well as the special link have to be created, adding to the overall cost. On the other hand, during the `compile` phase for instance, only one hash of the directory name results in the location of the physical node storing the file.

6.2. File Replication Overhead

In this section we evaluate the overhead of creating a replica in the system. For this purpose we used a five node setup with each node contributing 35 GB disk space, and chose to maintain three replicas of each stored file. To create a realistic file system, we collected a trace from the central NFS server of our department, and created similar files and directories under Kosha. The trace contained 221K files of 130 users, for a total of 17.9 GB of data. Once we allowed the system to be stable, we failed one node and determined the time it took for the system to create a new replica to compensate for the failed node. This represents the worst case scenario, as all files stored on the node would need to be copied. We repeated the experiment 5 times and determined that it takes on average 36.634 min to copy about an average of 12.2 GB of data. Hence, the cost of creating a replica is under an hour which shows that if the mean time to failure of a node is in the order of days (a reasonable assumption for the targeted system), the overhead introduced by the creation of new replicas is small. This reconfirms our notion that Kosha provides a practical approach that yields an economical fault tolerant distributed file system.

6.3. Load Distribution

The load distribution facilities of Kosha are evaluated in this section. For the purpose of these evaluations, we simulated a Kosha cluster of 16 nodes and fixed the number of replicas to 3. The simulation was driven by a file system trace with the same properties as the file system of Section 6.2.

The first set of experiments measured the effect of subdirectory distribution on the load balancing characteristics of the system. Each node contributed 10 GB of disk space to avoid file redirection. The distribution level was varied from 1 to 10, and for each level, we collected the distribution information from all nodes, and measured the number of files and their collective sizes on the individual nodes. The simulation was repeated 50 times varying the `nodeId` assignments in the Pastry network, and the results were averaged. We also calculated these quantities for a hypothetical

scheme which distributed individual files among different nodes. This finest grained distribution gives the upper bound on the best load balancing (for the trace used) that can be achieved using DHTs.

Figure 5 shows the result of the load balancing experiments. The dotted horizontal lines show the mean and the standard deviation of the distribution of the number and the collective size of files on the different nodes when each individual file was hashed and distributed. The results show that as the distribution level is increased, the load balancing in terms of the number of files converges toward the upper bound. The file size distribution improves with distribution level, but the improvement is not uniform because of the variance in the size of the directories being distributed. Using directory distribution with distribution level 4 or greater provides comparable load balancing to that of individually hashing all files.

The next set of experiments measured the effect of file redirection on the overall disk utilization. The simulation for this was done for a cluster of 16 nodes, 8 of which contributed 3 GB each, four nodes contributed 4 GB each, and four nodes contributed 5 GB each of disk space. These numbers were chosen to study the system under high utilization. The distribution level was fixed at 4, and the number of the replicas was fixed at 3. The file system trace from our department was once again used to drive the simulation, and the number of insertion failures was recorded as the files were added. The simulation was repeated with redirection attempts varying from 1 to 15. Each simulation was run 50 times varying the `nodeId` assignment in the Pastry network, and the results were averaged. In [27], the cumulative failure ratio is defined as the ratio of all failed insertions over all insertions that have occurred up to the point when the given storage utilization was reached. We use the same definition. Figure 6 shows the cumulative failure ratio *versus* the percentage utilization. It shows that with 4 redirection attempts and distribution level 4, the failure ratio remains near 0 for utilization as high as 60%, and it does exceed 12% when the utilization approaches 100%. Note that while increasing the number of redirection attempts results in a higher utilization of the total disk space, each redirection attempt requires hashing of the file name which can hinder the file operation performance.

6.4. Fault Tolerance

The experiments in this section measured the availability of Kosha under failures. We used an availability trace of 51, 663 machines in a large corporation over a consecutive 35-day (840-hour)

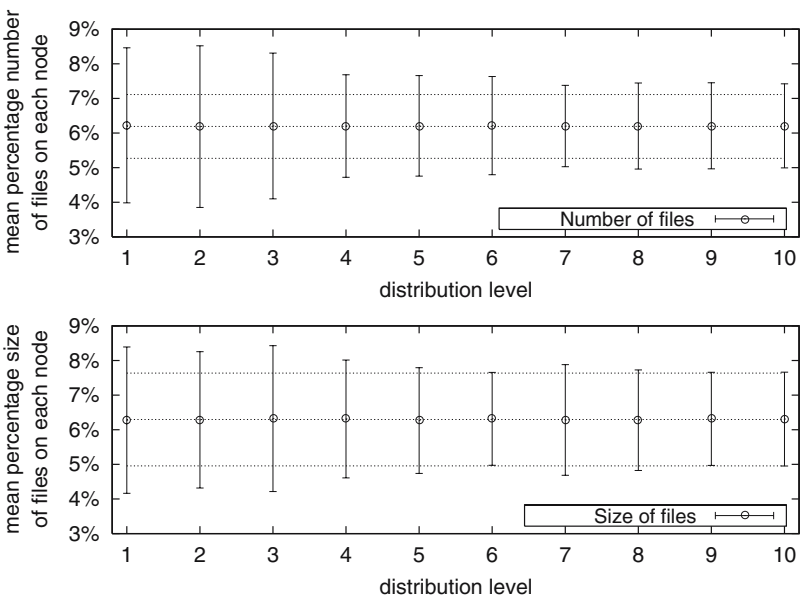

Figure 5 The mean and standard deviation of the percentage of the number of files and their sizes per node across 16 nodes as the distribution level is increased. The *dotted horizontal lines* show the mean and the standard deviation when each individual file was distributed to a different node, i.e., with the finest grained distribution.

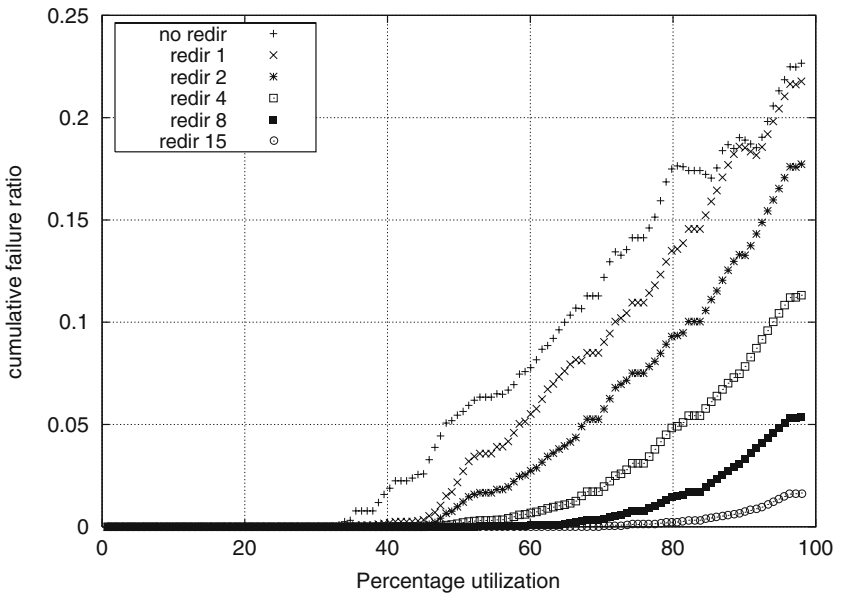

Figure 6 The cumulative failure ratio *versus* utilization, as the number of redirection attempts is increased. The distribution level is fixed at 4.

period [4]. The trace contains the status of machines (up or failed) recorded hourly. We simulated Kosha for the cluster of 51, 663 machines. We distributed the files obtained from the file system trace from Purdue University's servers as described earlier, and then used the availability data to introduce failures and node joins. For this experiment, we assume that the time to create a new replica of a node is 30 min. For each hour, we determined the total number of files that remain available. The distribution level was fixed at 3, and the experiments were repeated with the number of replicas varying from 0 to 4. For each case, 100 runs were made with various `nodeId`*s* for the nodes in the Pastry network, and the results were averaged.

Figure 7 shows the percentage of total files available over the 840 h period. The lower spike in the graph for Kosha-0, i.e., with no replicas, shows that the system performance is affected when a large number of failures occur. However, even maintaining a single replica (Kosha-1) increases the availability significantly, even for the case of a large number of simultaneous failures at hour 615. For the case of Kosha-3, the average availability is 99.991%, signifying that Kosha can guarantee near 100% availability with only three replicas. The reason for this is that Kosha continuously maintains the K replicas it was configured for (Section 4.2); node failures are tolerated as new replicas are created when old ones become unavailable.

7. Related Work

The main driving force behind widespread use of p2p techniques has been large-scale data sharing facilities such as Gnutella [18], Freenet [11], and Kazaa [30]. The basic data sharing is extended by providing strong persistence and reliability in p2p distributed storage projects, such as Pond [25] which is a prototype of Oceanstore [20], CFS [13], and PAST [27]. Kosha uses a similar p2p substrate but also provides a virtualized NFS interface that creates a file system abstraction to the distributed storage.

Scalable distributed [19] or serverless [3, 33] file systems provide some p2p aspects, but may not be practical to switch to in an established environment because of their fundamentally different designs and requirements.

There are several wide-area file system projects such as Ivy [23], Farsite [2], and Pangaea [28], which also provide reliability. In contrast to these file systems, Kosha does not focus on wide-area scalability. Instead, it focuses on extending the capabilities of a local-area NFS.

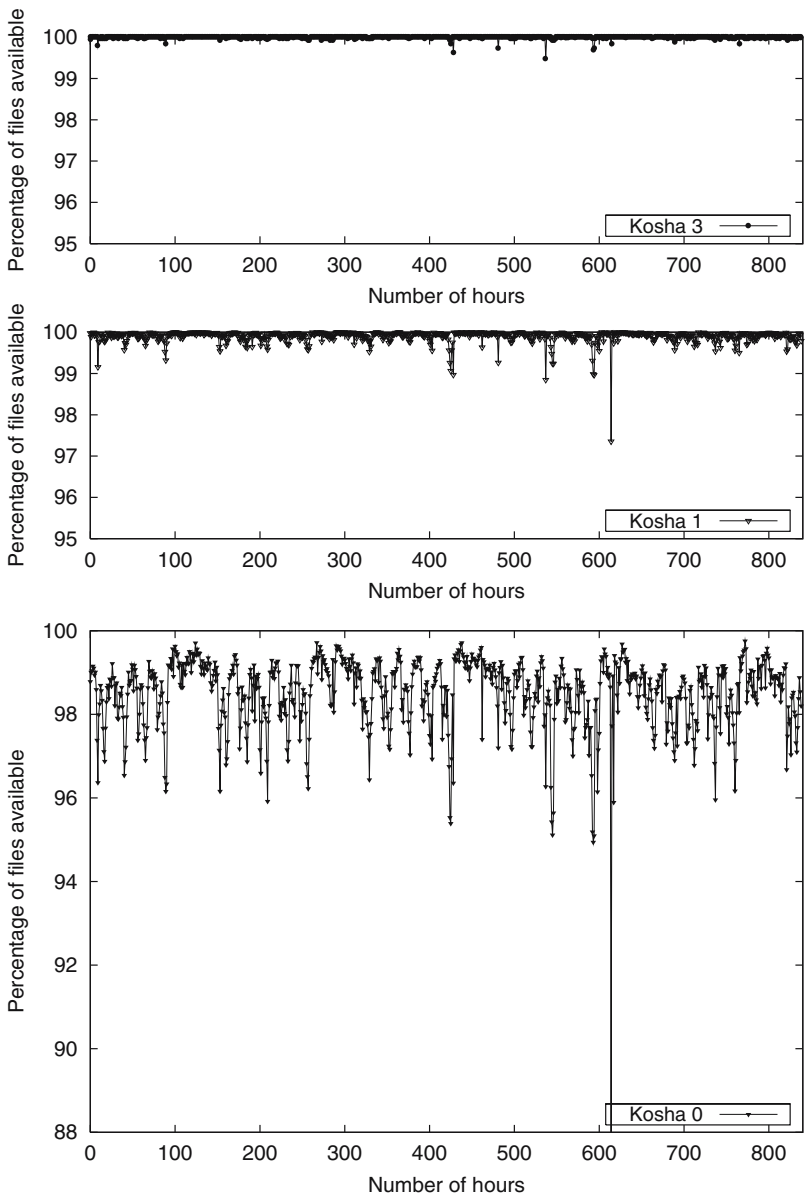

Figure 7 Percentage of total files that are available over a period of 840 h. The distribution level was fixed at 3 for these results. The largest number of failures (4, 890) occurred at hour 615, where over 12% files became unavailable for Kosha-0 compared to only 0.16% for Kosha-3.

Kosha is more likely to see actual use since wide-area file storage raises more issues of trust and consistency, despite the numerous approaches that have been developed to address these problems, such as encryption [11], agreement protocols [5, 10], and logs [23]. Kosha avoids most of these problems since it is only concerned with maintaining accurate replicas, while supporting standard NFS consistency semantics.

Finally, NFSv4 [31] also provides features of file system replication and migration. However, the replication is static. In NFSv4, a client first queries the main server which can provide it with a list of locations from where to obtain the files. The client then directs its queries to that location. In Kosha, this location is transparent and multiple queries are not required. Moreover, the goal of NFSv4 is to provide load balancing and fault tolerance, where as Kosha has an additional objective of utilizing unused disk space on cluster nodes and desktops. In the long run, Kosha can benefit from the file migration and replication facilities

provided by NFSv4 which can lead to a simplified design and a more efficient system.

8. Conclusion

We have presented Kosha, a p2p enhancement for the widely used NFS. By blending the strengths of NFS with those of p2p overlays, Kosha aggregates unused disk space on many computers within an organization into a single, shared file system, while maintaining normal NFS semantics. In addition, Kosha provides location transparency, mobility transparency, load balancing, and high availability through replication and transparent fault handling. Kosha effectively implements a 'Condor' [21] for unused disk storage.

We have built our Kosha prototype on top of the SFS toolkit [22], using the Pastry p2p overlay for node location in distributing directories. Performance measurements in a LAN show that Kosha over 16 nodes incurs a total overhead of 7.8%. Simulations using a large file system trace shows that Kosha's directory distribution techniques achieves a balanced load distribution similar to that of distributing individual files. Simulations using a machine availability trace collected in a large business organization show that Kosha guarantees near 100% availability during node failures by maintaining three replicas of each stored file. Since Kosha exports the NFS interface and consistency semantics, it is more likely to see actual use than techniques that provide fundamentally different interfaces.

References

1. F. 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington District of Columbia, April (1995)
2. Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In: Proc. OSDI, December (2002)
3. Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S., Wang, R.Y.: Serverless network file systems. ACM Trans. Comput. Syst. **14**(1), (1996)
4. Bolosky, W.J., Douceur, J.R., Ely, D., Theimer, M.: Feasibility of a serverless distributed system deployed on an existing set of desktop pcs. In: Proc. SIGMETRICS, June (2000)
5. Brodsky, D., Pomkoski, J., Feely, M., Hutchinson, N., Brodsky, A.: Using versioning to simplify the implementation of a highly-available file system. Technical Report TR-2001-07, The University of British Columbia, Canada, (2001)
6. Butt, A.R., Zhang, R., Hu, Y.C.: A self-organizing flock of Condors. In: Proc. ACM/IEEE SC2003: International Conference for High Performance Computing and Communications, Phoenix, AZ, November (2003)
7. Callaghan, B.: NFS Illustrated. Addison Wesley Longman, Inc., (2000)
8. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Rice Univeristy, (2002)
9. Castro, M., Ganesh, A., Rowstron, A., Wallach. D.S.: Security for structured peer-to-peer overlay networks. In: Proc. OSDI, December (2002)
10. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proc. OSDI, February (1999)
11. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. ⟨http://freenetproject.org/freenet.pdf⟩ (1999)
12. Compaq. Compaq Product Information. ⟨http://www.compaq.com/⟩ (2004)
13. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: Proc. SOSP, October (2001)
14. Dell Computer Corporation. Dell – Client & Enterprise Solutions, Software, Peripherals, Services. ⟨http://www.dell.com/⟩ (2004)
15. Druschel et al. Freepastry. ⟨http://freepastry.rice.edu/⟩ (2004)
16. Foster, I., Kesselman, C. Globus: A metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. High Performance Comput. **11**(2), 115–128, Summer (1997)
17. Foster, I., Kesselman, C. (eds.). The GRID: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, (1999)
18. Frankel, J., Pepper, T.: The Gnutella protocol specification v0.4. ⟨http://cs.ecs.baylor.edu/d~onahoo/classes/4321/GNUTellaProtocolV0.4Rev1.2.pdf⟩ (2003)
19. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.: Scale and performance in a distributed file system. ACM Trans. Comput. Syst. **6**(1), 51–81 (1988)
20. Kubiatowicz, J. et al.: Oceanstore: An architecture for global-scale persistent store. In: Proc. ASPLOS, November (2000)
21. Litzkow, M.J.M.J., Livny, M., Mutka. M.W.: Condor – A hunter of idle workstations. In: Proc. ICDCS, June (1988)

22. Mazieres, D.: A toolkit for user-level file systems. In: Proc. USENIX Technical Conference, June (2001)
23. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: A read/write peer-to-peer file system. In: Proc. OSDI, December (2002)
24. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A Scalable Content-Addressable Network. In: Proc. SIGCOMM, August (2001)
25. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: The oceanstore prototype. In: Proc. USENIX FAST, December (2003)
26. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proc. IFIP/ACM Middleware, November (2001)
27. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proc. SOSP, October (2001)
28. Saito, Y., Karamanolis, C., Karlsson, M., Mahalingam, M.: Taming aggressive replication in the Pangaea wide-area file system. In: Proc. OSDI, December (2002)
29. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun network file system. In: Proc. Summer USENIX, June (1985)
30. Sharman Networks. Kazaa Media Desktop. ⟨http://www.kazaa.com/index.htm⟩ (2004)
31. Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., Noveck, D.: RFC3530: Network File System (NFS) Version 4 Protocol. ⟨http://www.ietf.org/rfc/rfc3530.txt⟩ (2004)
32. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proc. SIGCOMM, August (2001)
33. Thekkath, C.A., Mann, T., Lee. E.K.: Frangipani: A scalable distributed file system. In: Proc. SOSP, October (1997)
34. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April (2001)