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

Ali Raza Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu
Purdue University
West Lafayette, IN 47907
{butta, troyj, yzheng, ychu}@purdue.edu

ABSTRACT

This paper presents Kosha, a peer-to-peer (p2p) enhancement for the widely-used Network File System (NFS). Kosha harvests redundant storage space on cluster nodes and user desktops to provide a reliable, shared file system that acts as a large storage with normal NFS semantics. 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 4.1% fixed overhead and 1.5% additional overhead as nodes are increased from one to eight. 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.

1. INTRODUCTION

This paper presents Kosha, a peer-to-peer (p2p) enhancement for the widely-used Network File System (NFS) [27, 6]. Kosha provides a single file system image identical to NFS, yet offers features commonly found in p2p storage systems [25, 12], such as location transparency, mobility transparency, load balancing, and high availability through file replication and transparent fault handling. Kosha leverages p2p technology and the unused disk space of desktop machines to enhance NFS. It does not entail changes to the underlying operating system, and requires only minimal configuration. 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.

The design of Kosha is aimed at academic and corporate networks on the order of 10^4 nodes, where NFS cross-mounting facilities are used extensively to provide users access to storage beyond their local disk. In such environments, efficiency and economics dictate the widespread use of off-the-shelf desktops for fulfilling the computing needs. These machines have become increasingly powerful, both in terms of processing power and storage capacity [16], and therefore have the potential to be used as shared resources if so desired. Unfortunately, a large amount of free disk space that exists on typical desktops is wasted as individual users are served mostly by central NFS servers, which must be upgraded to scale with the client population. Therefore a prime goal of Kosha is to utilize the cheap storage that is available in such environments, while achieving higher data availability and an acceptable level of load balancing.

It seems desirable to solve this problem, though likely it is not practical to replace NFS in an established computing environment (For similar reasons switching to AFS [17] or xFS [3] may not be possible.) Kosha addresses these issues, and provides additional features of fault tolerance and high availability, which come naturally from the use of a p2p overlay. Since the widespread use of NFS is indispensable in the targeted environments, *the key idea here is to extend NFS without incurring any changes to the underlying file system.* Specifically, Kosha organizes the nodes that contribute disk space into a structured p2p overlay which then uses NFS to replicate files across peers and make the location of the files 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 is built on top of NFS; it does not entail that nodes running Kosha cannot use standard NFS. The standard NFS is not affected, and the node owner can decide on the portion of a node's storage space that will be used for Kosha and standard NFS. Once this is done, the standard NFS and Kosha operations will not interfere with each other. 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 for users if the system administrators move all the users' files to Kosha mount points and set

ACM/IEEE SC '04 November 6-12, 2004, Pittsburgh, PA, USA
0-7695-2153-3/04 $20.00 ©2004 IEEE .

In *Proceedings of the ACM/IEEE SC2004: High Performance Computing, Networking and Storage Conference, Pittsburgh, PA, November 6-12, 2004.*

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. location transparency,
3. mobility transparency (i.e., transparent migration of files and subdirectories),
4. load balancing via an efficient scheme of distributing directories instead of files,
5. high availability through replication and transparent fault handling, and
6. 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 on which this work is based. 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 [13, 11]. 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 user 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[22], Chord[30], Pastry[24], and Tapestry[32] 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 $O(logN)$ overlay hops and each node maintains only $O(logN)$ 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 leveraged 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 [24, 7] 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,

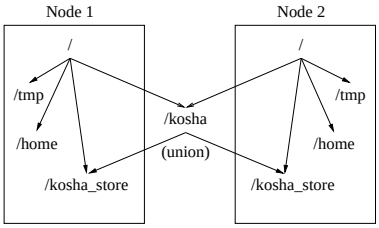

Figure 1: Virtual directory hierarchy: */kosha* is the virtual directory and is the union of */kosha_store* on all the nodes.

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 also 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 [24, 7].

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 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 */a/myFile* Kosha performs the following mapping:

$$/kosha/a/myFile$$

$$\Longrightarrow DHT(hash(a)) : /kosha\_store/a/myFile$$

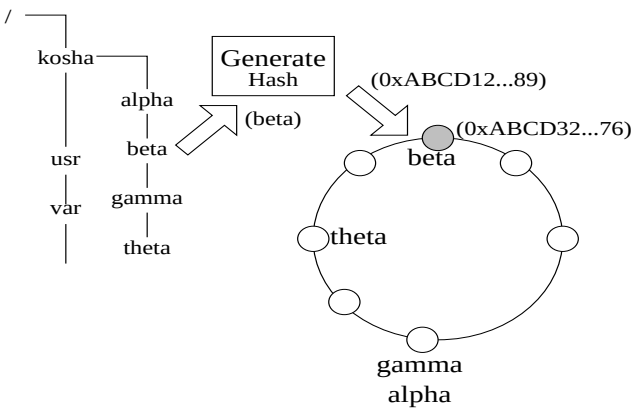

Figure 2: Example of file distribution to multiple nodes. The virtual mount point is */kosha*. The directory name is first hashed using the Generate Hash function to generate a unique key, which is then routed using Pastry to a node whose `nodeId` is numerically closest to the key. The selected Pastry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

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 [24], 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 directories are distributed to nodes other than the ones that store their parent directory, an empty special link as described in Section 3.3 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/a* and */kosha/b* may be stored on different nodes as determined by the p2p substrate, but the directories */kosha/a/x* and */kosha/a/y* are both stored on the same node as the one which stores */kosha/a*. Similarly, distribution level 2 implies that another level of subdirectories will also be distributed to multiple nodes. In this case, */kosha/a/x* and */kosha/a/y* may be stored on different nodes than the one that stores */kosha/a*. Hence, distribution level controls the granularity of distribution and load balancing.

3.3 Optimization

Because of the heterogeneity in the storage space contributed by nodes and variations in directory size, the node selected for storing a directory may not have enough local disk space to hold the directory and all its files. Since deciding the size of a directory a priori (i.e., without knowing the size of the files that will be created in it) is not possible, redirection is done for all newly created directories when the local disk space has exceeded the pre-specified utilization. When this happens, the directory is *redirected* and stored on a different node. Redirection is done by concatenating a random salt to the directory 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 [25].

When a directory is redirected, a special soft link to the redirected directory is created in the parent directory. A special link is a soft link that serves to connect the redirected directory to its actual location in the parent directory. The name of the link is the same as the name of the redirected directory; this helps Kosha list the directory contents of the parent directory. The target of the link is the directory name concatenated with the salt value. When Kosha comes across a special link, it follows the special link and accesses the redirected directory. 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 [24]. 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. At 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. Next, 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. The receiving node performs the operation and returns the results to Kosha, which then records the information needed for future accesses, and finally 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

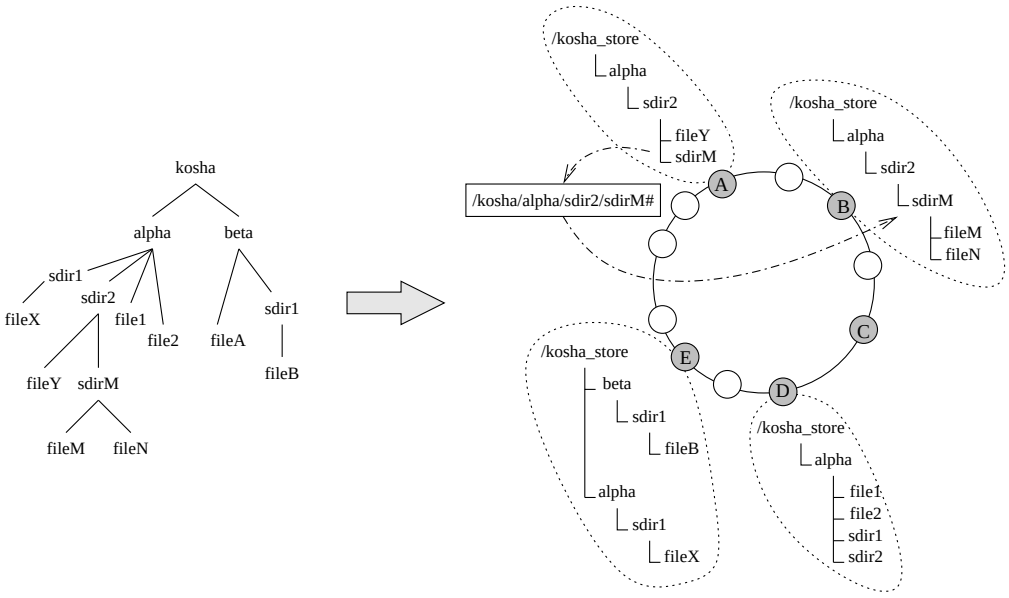

Figure 3: Example of subdirectory distribution to multiple nodes. The files in the same directory are stored on the same node as the parent directory. However, the subdirectories are distributed to remote nodes. The distribution level is set to 2, and A has limited capacity which causes */alpha/sdir2/sdirM* to be redirected. The example contents of the special link are shown in the rectangle.

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. In the case of distributed directories, subdirectories are also traversed and removed. The empty directory structure created to support the distributed subdirectory is then examined for possibly being used by other subdirectories with some common path prefix. The empty hierarchy leading to the subdirectory is then deleted, and the special link in the parent directory on the node to which the parent directory is mapped is also removed. 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 [9], as utilized in [26]. 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 [8]; 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.

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

The failure of a primary replica node is handled transparently. For the purpose of this discussion we assume only crash failure. 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` in the root directory of the replicated hierarchy, 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.

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 [20] 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 applica-

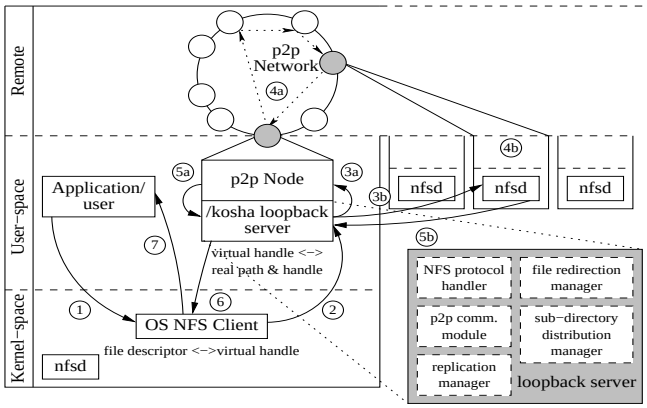

Figure 4: Kosha architecture: (1) application makes an I/O system call, (2) kernel makes an RPC call, (3) (a) local port request to peer substrate or (b) handle substituted and RPC forwarded, (4) (a) overlay locates node storing file or (b) file I/O occurs, (5) (a) local port reply from peer substrate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

tion 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).

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 available version of this API is FreePastry [14] 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 [20] with 4000+ 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) [26] and compared it to NFS Version 3. These experiments were performed on an 8-node configuration. Each node has a 2.0 GHz Intel P4 with 512 MB RAM and a 40 GB 7200 RPM Barracuda Seagate hard disk, and runs FreeBSD 4.6. The nodes are connected via a 100Mb/s Ethernet switch. The file distribution used is 51MB in size, with a maximum subdirectory level of 4.

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

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, 50 runs of the benchmark were made, and the execution time for each phase was recorded. For Kosha, we measured the performance as we successively increased the number of nodes from 1 to 8. The NFS configuration consists of two nodes with one running as a client, and the other running as a server.

The total overhead introduced by Kosha, as compared to the performance of NFS, is under 6%. Adding more nodes into the system does not affect the overall performance drastically (only 1.5% additional total overhead introduced when the number of nodes increased from 1 to 8), 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 does not exceed 4ms plus a constant factor. This shows that Kosha is highly scalable; additional nodes can be introduced into the system with little impact on the performance.

Table 1: Performance of a modified Andrew benchmark on Kosha with increasing number of nodes. The table shows execution times of each phase in seconds and respective overhead compared to NFS. The distribution level for Kosha was fixed at 1 for these measurements.

Benchmark	NFS	Kosha							
		One Node		Two Nodes		Four Nodes		Eight Nodes	
		exec. time	ovrhd	exec. time	ovrhd	exec. time	ovrhd	exec. time	ovrhd
mkdir	0.201	0.203	1.010	0.205	1.020	0.207	1.030	0.208	1.035
copy	24.644	25.834	1.048	25.872	1.050	25.888	1.050	26.101	1.059
stat	0.594	0.711	1.197	0.756	1.273	0.761	1.281	0.810	1.364
grep	2.803	2.829	1.009	2.899	1.034	2.901	1.035	2.954	1.054
compile	27.418	28.391	1.035	28.510	1.040	28.515	1.040	28.712	1.047
Total	55.660	57.968	1.041	58.242	1.046	58.272	1.047	58.785	1.056

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 4. 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	0.207	0.232	1.12	0.245	1.18	0.248	1.20
copy	25.888	28.324	1.09	28.943	1.12	29.010	1.12
stat	0.761	0.930	1.22	0.977	1.28	0.981	1.29
grep	2.901	3.012	1.04	3.101	1.07	3.143	1.08
compile	28.515	28.952	1.02	30.100	1.06	30.487	1.07
Total	58.272	61.450	1.05	63.366	1.09	63.869	1.10

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 4. Once again, 50 runs of the MAB were made, and the execution times were recorded.

Table 2 shows that the overhead in distribution levels 2, 3, and 4 relative to distribution level 1 are 5%, 9%, and 10%, respectively. This implies that having a large distribution level is not inhibiting. Also observe that the cost on `mkdir` and `copy` 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 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, which we collected from the central NFS server of our department. The trace contained 221K files of 130 users, for a total of 17.9 GB of data.

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, but the improvement is not monotonic. This is because the distribution process is not based on file sizes. 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, 4 nodes contributed 4 GB each, and 4 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

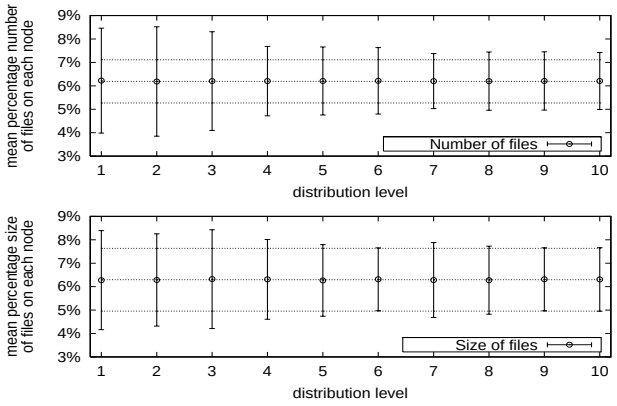

Figure 5: The mean and standard deviation of the percentage of the number of files and their sizes per node across 16 nodes as the distribution level is increased. The dotted horizontal lines show the mean and the standard deviation when each individual file was distributed to a different node, i.e.,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

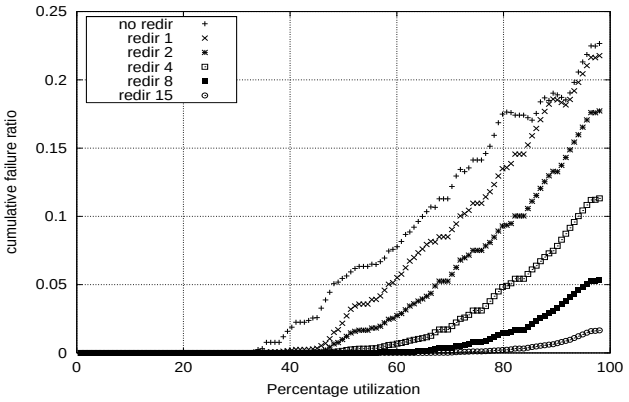

Figure 6: The cumulative failure ratio versus utilization, as the number of redirection attempts is increased. The distribution level is fixed at 4.

the number of insertion failures was recorded as the files were added. The simulation was repeated with file 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 [25], the cumulative failure ratio is defined as the ratio of all failed file insertions over all file 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.

6.3 Fault tolerance

The experiments in this section measured the availability of Kosha under failures. We used an availability trace of 51663 machines in a large corporation over a consecutive 35-day (840-hour) period [4]. The trace contains the status of machines (up or failed) recorded hourly. We simulated Kosha for the cluster of 51663 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 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 hours 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 [15], Freenet [10], and Kazaa [28]. The basic data sharing is extended by providing strong persistence and reliability in p2p distributed storage projects, such as Pond [23] which is a prototype of Oceanstore [18], CFS [12], and PAST [25]. 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 [17] or serverless [3, 31] 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 [21], Farsite [2], and Pangaea [26], 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.

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 [10], agreement protocols [9, 5], and logs [21]. Kosha avoids most of these problems since it is only concerned with maintaining accurate replicas, while supporting standard NFS consistency semantics.

Finally, NFSv4 [29] 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

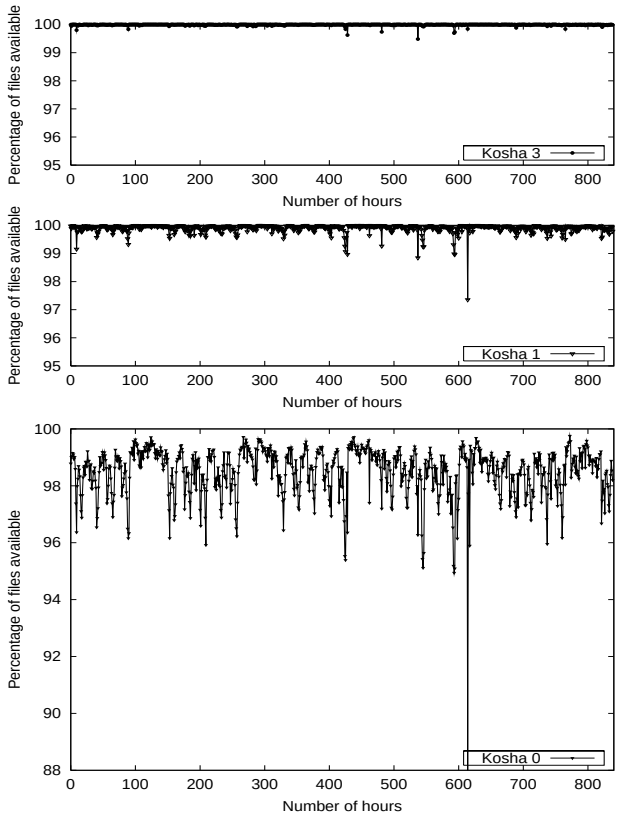

Figure 7: Percentage of total files that are available over a period of 840 hours. The distribution level was fixed at 3 for these results. The largest number of failures (4890) occurred at hour 615, where over 12% files became unavailable for Kosha-0 compared to only 0.16% for Kosha-3.

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" [19] for unused disk storage.

We have built our Kosha prototype on top of the SFS toolkit [20], using the Pastry p2p overlay for node location in distributing directories. Performance measurements in a LAN show that Kosha over eight nodes incurs a total overhead of 5.6%. 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.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. 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.

9. REFERENCES

[1] F. 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington D.C., April 1995.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. OSDI*, December 2002.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1), February 1996.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed system deployed on an existing set of desktop pcs. In *Proc. SIGMETRICS*, June 2000.

[5] D. Brodsky, J. Pomkoski, M. Feely, N. Hutchinson, and A. Brodsky. 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] B. Callaghan. *NFS Illustrated*. Addison Wesley Longman, Inc., 2000.

[7] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Rice Univeristy, 2002.

[8] M. Castro, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proc. OSDI*, December 2002.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, February 1999.

[10] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. ⟨ http://freenetproject.org/freenet.pdf ⟩ (1999).

[11] Compaq. Compaq Product Information. ⟨ http://www.compaq.com/ ⟩ (2004).

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, October 2001.

[13] Dell Computer Corporation. Dell - Client & Enterprise Solutions, Software, Peripherals, Services. ⟨ http://www.dell.com/ ⟩ (2004).

[14] Druschel et. al. Freepastry. ⟨ http://freepastry.rice.edu/ ⟩ (2004).

[15] J. Frankel and T. Pepper. The Gnutella protocol specification v0.4. ⟨ http://cs.ecs.baylor.edu/~donahoo/classes/4321/GNUTellaProtocolV0.4Rev1.2.pdf ⟩ (2003).

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (3rd Edition)*. Morgan Kaufmann Publishers, 2002.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[18] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS*, November 2000.

[19] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. ICDCS*, June 1988.

[20] D. Mazieres. A toolkit for user-level file systems. In *Proc. USENIX Technical Conference*, June 2001.

[21] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. OSDI*, December 2002.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. SIGCOMM*, August 2001.

[23] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proc. USENIX FAST*, December 2003.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[25] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, October 2001.

[26] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. OSDI*, December 2002.

[27] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Summer USENIX*, June 1985.

[28] Sharman Networks. Kazaa Media Desktop. ⟨ http://www.kazaa.com/index.htm ⟩ (2004).

[29] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC3530: Network File System (NFS) Version 4 Protocol. ⟨ http://www.ietf.org/rfc/rfc3530.txt ⟩ (2004).

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. SIGCOMM*, August 2001.

[31] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. SOSP*, October 1997.

[32] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.