On Utilization of Contributory Storage in Desktop Grids

Chreston Miller, Ali R. Butt, and Patrick Butler
Department of Computer Science, Virginia Tech
Blacksburg, VA 24061
{chmille3, butta, pabutler}@cs.vt.edu

Abstract

Modern desktop grid environments and shared computing platforms have popularized the use of contributory resources, such as desktop computers, as computing substrates for a variety of applications. However, addressing the exponentially growing storage demands of applications, especially in a contributory environment, remains a challenging research problem. In this paper, we propose a transparent distributed storage system that harnesses the storage contributed by desktop grid participants arranged in a peer-to-peer network to yield a scalable, robust, and self-organizing system. The novelty of our work lies in (i) design simplicity to facilitate actual use; (ii) support for easy integration with grid platforms; (iii) innovative use of striping and error coding techniques to support very large data files; and (iv) the use of multicast techniques for data replication. Experimental results through large-scale simulations, verification on PlanetLab, and an actual implementation show that our system can provide reliable and efficient storage with support for large files for desktop grid applications.

1. Introduction

In recent years, the desktop computer has become a powerful resource that has the capability to support far more complex and demanding applications than those of a typical desktop user. This advancement has paved the way for large-scale distributed computing systems based on desktop machines referred to as desktop grids. As more and more efficient desktop grid systems such as Condor [19] and Entropia PC Grids [6] are being designed and deployed, their use as resource providers for modern scientific applications is becoming increasingly popular [15, 18].

While the focus of desktop grids has mainly been on providing computational resources to execute user submitted jobs, e.g., Condor [19], addressing the ever-increasing storage demands of applications has largely been ignored. Multimedia files, high-resolution medical images, weather forecast data, and virtual environment data for human-computer interaction applications are just a few of the examples of large files that can be processed using desktop grid resources. The existing I/O model of storing all the application input/output files on either the job submission machine, e.g., as in Condor [19], or copying between the submission and execution machines, e.g., as in Globus [11], implies that the submission as well as the execution machine should have the capacity to store the required files in their entirety, or the application is explicitly aware of the distributed locations of all the data it will access [5]. The large size and dynamic nature of data used by modern grid applications [33] implies that neither limiting the size of the data by available space on a single machine, nor explicitly specifying data location, is a feasible approach.

Recently, a number of distributed storage systems [2, 9, 23, 29, 31, 35] have leveraged peer-to-peer (p2p) overlay networks to provide scalability, self-organization, and reliability. These systems have shown that p2p networks can serve as a suitable communication substrate for large-scale storage applications. While the issues of distribution, location, replica management, and fault-tolerance are discussed in varying details in these systems for a variety of target environments, these systems either do not address how large data files can be stored, or they rely on complex solutions that result in non-standard interfaces. This makes an easy adaptation of such storage systems into today's desktop grids an uphill battle.

In this paper, we develop a p2p storage system, PeerStripe[1], that provides an economical and efficient storage solution for large data files. Our goal is an elegant and simple system design that allows for files to be stored on participating nodes that have joined a p2p overlay network. Our use of p2p networks ensures that the proposed system can support the features of scalability, self-organization, reliability, and composability for target environments of various sizes. A unique feature of PeerStripe is that instead of storing entire files on individual nodes, it splits the files into varying sized chunks and then stores these chunks sep-

[1] An initial version of PeerStripe was explored in [22].

arately on heterogeneous nodes distributed across a wide-area network. This approach is inspired by the data striping techniques employed in local-area RAID [25] clusters. As a result, unlike previously proposed approaches such as PAST [31], the size of a file that can be stored in PeerStripe is not limited by the capacity of an individual participating node. Moreover, to protect against losing data due to losing a chunk of a distributed file, we employ error coding at the granularity of the chunks. Error coding also ensures that PeerStripe provides fault-tolerance and data availability despite churn of system participants.

Users and applications can access the distributed storage exported by PeerStripe either explicitly by using its API that allows storing and retrieval of entire as well as portions of files, or implicitly by linking an application with the PeerStripe library that intercepts application I/O and performs the necessary redirection. In this way, our system can easily be interfaced with existing as well as new applications. PeerStripe supports transparent distribution, striping, and look up of data files across participating nodes, and hence can serve as robust and easy-to-use storage for desktop grids.

The main contributions of this paper are as follows:

1. A simple yet efficient storage system design that supports storing large data files on participants in a structured p2p network, and provides a rich set of features such as mobility and location transparency, self-organization, load-balancing, and decentralized operation;

2. An innovative adaptation of striping and data error coding techniques to provide fault tolerance in a wide-area p2p-based distributed storage system;

3. An exploration of multicast techniques for data replication;

4. An implementation that allows easy integration with applications; and

5. A detailed evaluation via large-scale simulations, verification on the PlanetLab [26] testbed, and an implementation study of how the system can be interfaced with Condor [19].

The rest of the paper is organized as follows. Section 2 presents a survey of related work and describes the building blocks used in the design of our system. Section 3 gives the motivation for our design. Section 4 presents the system design. Section 5 describes our implementation. Section 6 presents the evaluation of PeerStripe, and finally Section 7 concludes the paper.

2. Survey of Related Work

The design of PeerStripe is based on the observation that typical desktop machines in academic and corporate settings have a large amount of unused disk space [14, 35]. We assume that the owners of the machines are willing to share their unused storage space along with their computational resources as part of a desktop grid environment. These assumptions are in line with those made by other resource sharing systems [6, 11, 19, 31, 35].

In the following sections, we discuss related work and technologies that serve as building blocks for PeerStripe.

P2P-based storage Structured p2p overlay networks such as Chord[32], CAN[28], Pastry[30], and Tapestry[36] essentially provide a *distributed hash table* (DHT) abstraction. Each node in a structured p2p network has a unique node identifier (`nodeId`) and each data item stored in the network has a unique key. The `nodeIds` 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 DHTs have been successfully used for distribution of files on participating nodes in p2p-based storage systems such as PAST [31], CFS [9], Pond [29], IVY [23], and FARSITE [2]. These systems provide strong persistence and reliability, and are complimentary to the design of PeerStripe. Next, we discuss in more detail two such systems that we have used in our evaluation.

PAST [31] is a large-scale, Internet-based, storage utility, which uses the p2p network provided by Pastry [30] as a communication substrate. PAST provides scalability, high availability, persistence and security. Any online machine can act as a PAST node by installing the PAST software, and joining the PAST overlay network. A collection of PAST nodes forms a distributed storage facility, and store a file as follows. First, a unique identifier for the file is created by performing a universal hashing function such as SHA-1 [1] on the file name. Next, this unique identifier is used as a key to route a message to a destination node in the underlying Pastry network. The destination node serves as the storage point for the file. Similarly, to locate a file, the unique identifier is created from the file name, and the node on which the file is stored is determined through Pastry routing. PAST utilizes the excellent distribution and network locality properties inherent in Pastry. It also automatically negotiates node failures and node additions. PAST employs replication for fault tolerance, and achieves load-balancing among the participating nodes. Our work builds on the functions provided by PAST to store and retrieve portions of file, and adapts the core PAST functions to handle large files.

CFS [9] provides a scalable, wide-area storage infrastructure for content distribution. CFS exports a file system (hierarchical organization of files) interface to clients. It distributes a file over many servers by chopping every file into small (8 KB) blocks thereby solving the problem of load balancing for the storage and the retrieval of popular big files. This also results in higher download throughput for big files, which can be retrieved in parallel from many nodes. The component that stores data is referred to as a publisher. A publisher identifies a data block by a hash of its contents, and also makes this hash value known for others. Similarly, a client uses the identifier hash of a block and Chord [32] routing to locate and retrieve the block. To ensure authenticity of retrieved data, each block is signed using the publisher's well known public-key. Also, to maintain data integrity, blocks can only be updated by their publishers. Finally, CFS deals with fault tolerance by replicating each data block on k successors, where one successor is made in charge of regenerating new replicas when existing ones fail.

In contrast to the p2p-based storage systems, the Google File System (GFS) [12] employs a hierarchical scheme to provide large-scale storage at high-speeds. The files are statically split into chunks, and distributed to multiple *chunkservers* for storage. GFS employs large-scale replication to ensure data integrity under failures, and the overall GFS design is optimized for append writes. However, the design of GFS is limited to a single organization where all resources are owned/controlled by one entity. GFS cannot be simply extended to work across organizations which is required for contributory storage.

Another participant-based system is LOCKSS [20], that is aimed at providing storage for large data sets. The key idea is to use participating computers as back-up locations for files in case the main computers that are serving the content fail. This system is similar to ours in that it utilizes unused storage on remote participant nodes. However, LOCKSS is targeted at storing digital publications, which are typically much smaller in size compared to the application data targeted in this work.

Finally, systems such as Kosha [35] and TFS [7] provide transparent access to p2p-storage. Kosha provides a Network File System interface to a p2p storage system, and allows users and applications to transparently access their distributed files using a virtual directory hierarchy. This work differs from Kosha in two main aspects. First, the maximum size of a file that can be stored using Kosha is limited by the capacity of an individual participating node, whereas PeerStripe employs striping and error coding techniques to allow storage of very large files. Second, PeerStripe allows users to utilize the system either implicitly by requiring to link with special libraries or explicitly by using the `insert` and `lookup` API. This implies that no special administrative privileges are required to use PeerStripe, which is in contrast to Kosha, where administrative privileges are required for the setup. Hence, PeerStripe's approach is more portable and easier to use.

The focus of TFS [7] is on how to contribute maximum disk space with the least effect on the file system in terms of performance and capacity. The key is that TFS doesn't require the host OS to keep track of which blocks are contributed. The host OS can overwrite any block at any time and TFS keeps track of each block's status. TFS comprises of an in-kernel file system and a user space tool, `setpri`, for designating files and directories as either transparent or opaque.

Our system shares with these works the goal of using peer nodes to establish a participant-based contributory storage facility, but differ in that our work targets transparently providing storage for grid applications, utilizes a simple and effective design, and focuses on how large data files can be efficiently stored. We do not aim to provide a general-purpose file system but rather a distributed storage facility that can be easily integrated into grid applications, and, in that, avoid the overhead and complexity of supporting a distributed file system abstraction.

Erasure codes The techniques of striping and error coding used in our system are the hallmark of RAID [25], which uses several storage devices in parallel to provide reliable storage for files. However, RAID is generally used in local storage devices, which are identical, fixed in number, and have low failure rates. We extend and adapt these concepts to a wide-area distributed setting where nodes are heterogeneous and highly dynamic.

In order to ensure availability of data when nodes fail, we utilize erasure codes. In general, erasure codes break a message or chunk of data into several blocks (n) and encode each block. Due to the addition of redundancy information, the size of the encoded block is greater than the original block. Thus, encoding of n blocks results in $(n + k)$ encoded blocks, where k is an overhead due to the redundancy information for all the n blocks and depends on the kind of erasure code used. The goal is to support recovery of the original data given a partial subset of the $(n+k)$ blocks [27].

Recently, a new class of sub-optimal erasure codes, called online code [21, 27] have been proposed. Online code allow creation of as many blocks of encoded data as necessary (not limited to $(n + k)$ as before) for a given environment, but still supports data decoding using a much smaller subset of the total encoded blocks. Online code exhibits $O(1)$ encode time and $O(n)$ decode time per block. In the context of our system, the online code has the additional advantage that if nodes storing some of the encoded blocks fail, new encoded blocks can be created without loss of data. Such re-creation of encoded data entails a processing overhead. However, online code allows encoded blocks to be decoded independently and simultaneously, which implies that a significant portion of the block re-creation overhead can be hidden from the user by overlapping the re-creation process with retrieval and decoding of other blocks.

Data transfer using multicast A number of systems such as Bullet [17], Shark [3], and CoBlitz [24] have explored the use of multicast and p2p-techniques for transferring large amounts of data between a source and a destination. Large Internet data transfers are also explored by BitTorrent [8], which divides a file into chunks and then each chunk is replicated a large number of times to allow for fast downloads from different replicas. Glacier [13] provides a massive storage system and handles correlated failures of replicas. Finally, IBP [4] leverages strategic data placement to support faster file downloads and to store sensor data, e.g., Network Weather Service [34] data, close to the sensors for improving data access times. Inspired by these systems, we have investigated data replication using multicast techniques similar to those of Bullet.

3. Motivation

While current p2p storage systems provide a number of features necessary for utilizing them in a desktop grid environment, we observed several shortcomings: maximum size of data files that can be stored in the system limited to storage capacity of individual contributors [31]; use of simple replication to k replicas, which only provides reliability against k simultaneous failures [9, 31, 35] and wastes storage space if k is set too large; and supporting large files by dividing them into fixed size chunks [9], which results in scalability issues as the chunks per file increase proportionally to the file size. This work aims to address some of these challenges, in particular the handling of large data files.

Several systems such as CFS [9] store large data files using a shared pool of storage resources by dividing files into fixed size chunks. However, dividing the file into fixed size chunks poses a hurdle to the performance and utility of the system. In systems that do not split stored files, e.g., PAST [31], only a single p2p message is required to locate the participant that stores a file. In contrast, for CFS the number of such messages is proportional to the number of chunks and hence the size of the file. This implies that CFS is unlikely to efficiently scale with the size of files.

A motivation for using fixed size chunks is that given the small size of a chunk compared to the file, the probability to find a node that can store a chunk is higher than that for the entire file. However, we note that due to the heterogeneous storage capacities of the nodes, some nodes (E's) will have little capacity left even if the overall system utilization is low. Let the probability of a store to fail because it is mapped to E be p. Then the probability of a store to fail in PAST is simply p, and PAST addresses this problem by incorporating a retry mechanism that essentially rehashes the file name with a new salt value and repeats the p2p lookup procedure. Now, lets assume that p remains unchanged during the store of all the chunks of a file in CFS. Then in a simple scenario without any replication, the probability that the store of a file with n chunks will fail is given by $1-(1-p)^n$. This probability of failure is clearly very high, e.g., for a very lightly utilized system with $p = 0.1\%$, a store of 4 GB file using a chunk size of 4 MB has a failure probability of 64.1%, which increases to 98.3% for a 16 GB file. CFS does incorporate a retry mechanism per chunk, but that does not reduce the number of chunks, and hence the above discussed problem remains. Because of such scalability issues with using fixed-sized chunks, we investigate the use of varying chunk sizes to alleviate these problems, and therefore aim to provide robust support for large data files.

4. Design

In this section, we present the design of PeerStripe. For the following discussion, we refer to the machines that participate in our system as "nodes".

4.1. Overview

The first task of PeerStripe is to establish a pool of shared storage resources. We accomplish this by using the decentralized and robust communication substrate provided by Pastry [30] to arrange the nodes in a p2p overlay network. Once nodes become part of the overlay, they can reach each other and utilize and contribute to the shared storage.

A key feature of our design is to provide storage for large files whose size is larger than the capacity of any individual node. For this purpose, we split a file into chunks, and store the chunks in the storage pool. When it is desired to retrieve a file, all the chunks making up the file are located and assembled together. An advantage of splitting files is that the system does not have to retrieve an entire file if only a portion of the file is accessed, rather, only the chunk(s) containing that portion are retrieved. However, a possible problem is that the loss of a chunk of a file due to node failures may result in the entire data in the file becoming useless. We employ erasure codes to address this issue and to provide fault tolerance.

To manage storing and retrieval of chunks, we utilize Pastry's DHT abstraction of the nodes to map the chunks to nodes. To store a chunk from a node *S*, a unique identifier (UID) for the chunk is first calculated by performing a SHA-1 hash on the chunk name. The UID is then used as a key to send out a `lookUp` message in the overlay. The

```
(1)  0,5242880
(2)  5242881,26083328
(3)  26083329,52297728
(4)  52297729,86114304
(5)  86114305,86114304
(6)  86114305,104856576
```

Figure 1. Example contents of a CAT file. Each line represents a chunk. The total size of the file is about 100 MB. Chunk #5 is empty.

DHT guarantees that the message will be received at some target node *T* in the overlay. Upon receipt of the `lookUp` message, *T* replies with an acknowledgment message that contains the IP address of *T*. When *S* receives the acknowledgment from *T*, the instance of PeerStripe on *S* concludes that the chunk should be stored on *T*. Note that the actual store of the chunk is done directly over the IP network and does not involve the overlay. Similarly, to retrieve a stored chunk, a `lookUp` message is used to determine the target node that stores the chunk, and the actual retrieval is done over the IP network.

We note that most scientific data in the target environment is immutable after it is recorded, e.g., climate observations, or created, e.g., output of high-energy particle physics simulations. For this reason, our design focus on storage and preservation of immutable data, and does not concern with issues such as maintaining consistency among various replicas, and node-level issues of cache consistency between local cache and remote file data. We do however support append writes, which are supported at the granularity of chunks so that chunk rewriting and associated consistency issues can be avoided.

4.2. Chunk Storage and Error Coding

A file is stored in PeerStripe as follows. The file is first split into chunks. Each chunk is then divided into n blocks of equal size and error coded to give m encoded blocks, which are also of the same size. Next, instead of storing the original chunks as described in the previous section, we store the encoded blocks in the shared storage pool. The storing process is similar to that described for chunks earlier.

The size of chunks is not fixed and can vary, which raises the issue that there is no direct mapping between a file offset and the chunk that stores the offset. This is remedied by maintaining a chunk allocation table. Each row in this table represents a chunk and lists the portion of the file contained in that chunk expressed as minimum and maximum byte offset values. PeerStripe creates the chunk allocation table when a file is stored, and stores it in the p2p storage under the name *filename.CAT*. Figure 1 shows an example *CAT* file.

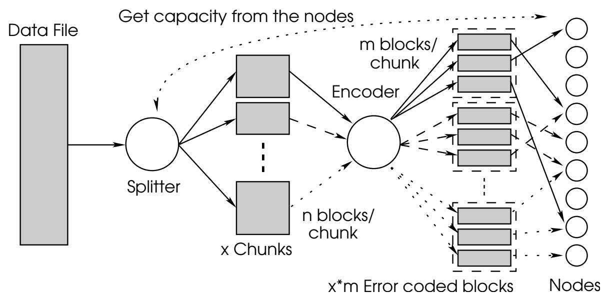

Figure 2. The various steps of storing a file in PeerStripe.

Retrieval of an entire file or a portion of the file involves the following sequence of events. PeerStripe first retrieves the associated *CAT* file by doing a DHT lookup for *filename.CAT*, and determines the number of the chunk to retrieve. Next, enough blocks are retrieved to allow decoding of the chunk. The process is repeated until the desired number of chunks is decoded. These chunks are then assembled into the file and returned to the user. For example, to retrieve an entire file *myTestFile* that contains three chunks under an XOR coding scheme that requires two encoded blocks to decode a chunk, PeerStripe will locate the encoded blocks: $myTestFile\_x\_y; 0 \leq x < 3, y$ any two in $\{0, 1, 2\}$.

4.3. Determining Chunk Sizes

Our design supports varying size chunks, with the goal to reduce the total number of chunks per-file. For this purpose, the node on which a chunk will be stored is queried for the size of a chunk it is willing to store. The size specified by the remote node is determined by its local policies, current load, and disk I/O performance, and can also be zero, which indicates that the remote node is either out of space or unwilling to store data. If this happens, the system can simply treat the current chunk as a chunk of size zero, and continue normally. In addition, PeerStripe factors in a node's performance in terms of available network bandwidth from the node in deciding the size. In the current implementation, PeerStripe simply does not use a node if the available bandwidth varies more than a pre-specified threshold[2]. Once a size is determined, a chunk of that size is created from the file. The chunk is then error coded and stored as discussed in the previous section. The process is repeated until all the data in the file is stored. Figure 2 illustrates various steps of this process.

[2]More advanced techniques for estimating a node's failure rate such as relying on Network Weather Service [34] can also be employed, and are the focus of our future work.

The advantage of using varying size chunks is that the number of chunks are dependent on the capacity of the system and not the length of the file being stored. Moreover, a system of retries to guard against failures is built in by allowing chunks to be of zero size. We do limit the number of consecutive zero-sized chunks in a file to protect against unbounded retries in case the system utilization is high. If this limit is exceeded, the file store fails and an error is returned to the user.

4.4. Fault Tolerance

The primary means of fault tolerance in our system is error coding. As nodes fail, the error coded blocks stored on them are lost and should be re-created to maintain redundancy. For this purpose, we leverage the Pastry leaf-set that maintains information about a node's neighbors in the identifier space, and Pastry's ability to detect a failure of a neighbor. Moreover, in Pastry the identifier space that is mapped to a failed node is split between the two immediate left and right neighbors of the failed node. This implies that a node whose immediate neighbor has failed becomes responsible for storing some of the blocks originally stored on that neighbor. Each node in our system has a list of blocks stored on its neighbors, and this list is updated when files are created or removed. When an immediate neighbor of a node fails, the node examines the list of blocks and determines which of these blocks will now be mapped to itself. For these blocks, the node starts the process of re-creating the lost encoded blocks using the remaining encoded blocks. Since we employ online code, a re-created block does not have to be exactly the same as the one that has been lost, rather functionally equal to support correct decoding of the stored data.

An interesting problem arises when a node that stores a large number of chunks fails, and its neighbors may not have the capacity to take over and store those chunks. This can possibly be avoided through our use of online code that allows us to simply drop, i.e. not recreate, an encoded chunk on a neighbor node, and create another one at a different location.

Finally, we also employ replication for the *CAT* file associated with each stored file. Note that the *CAT* file can be recreated by incrementally looking up chunks of a file and determining their size, however, given that active replication is in place, such recreation is expected to be rare.

Managing Replicas In addition to error coding, we have also employed simple replication of encoded blocks on neighboring nodes in the Pastry `nodeId` space. Instead of choosing a primary node and making it responsible for creating replicas as is the case in many systems [9, 31, 35], we utilized a multicast scheme to simultaneously create k replicas.

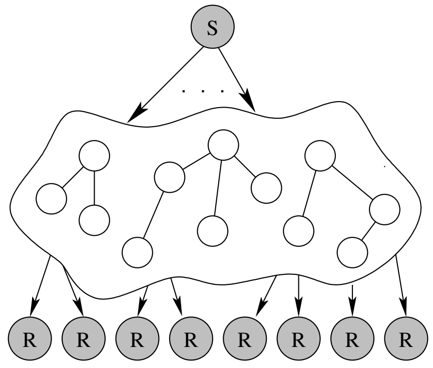

Figure 3. An example multicast tree structure for simultaneously creating replicas on nodes R of a chunk from source S.

Once a target node has been selected for storing an encoded chunk using the p2p-mapping, we determine $k-1$ of its neighbors in the identifier space and then leverage Bullet [17] to construct an overlay tree with the node starting the store as the source and the k selected nodes (the target node and its neighbors) as the leaf nodes. This is illustrated in Figure 3.

The challenging task is the creation of an effective tree. This can be achieved if a child is as physically close to its parent as possible. We leveraged the proximity-aware routing table of Pastry to realize this tree. Starting from the source node, we picked K closest nodes from the routing table as children, and then continue this step at each child as we moved towards the identifiers of the target nodes. As a result, the desired locality-aware tree is created. Note that our greedy approach does not guarantee that the overall tree follows the shortest path from the source to the destination, but it does provides strong locality at each step. Once the tree is created, we use the Bullet algorithm to multicast the data to the k replicas.

4.5. Discussion

The design of our system results in dividing large files into relatively few chunks. However, a number of systems [3, 24] have shown that having a file distributed across a number of nodes (a large number of chunks in the terminology of this work) can provide better transfer bandwidth when accessing the stored data. So, while large chunk sizes can provide easy location and reduce p2p-lookup overhead, smaller chunk sizes can provide better transfer bandwidths if portions of files are accessed in parallel by different nodes, and also entail faster regeneration of a lost chunk because of its smaller size. This leads to trade-offs in the selection of lower and upper bounds for chunk sizes. While,

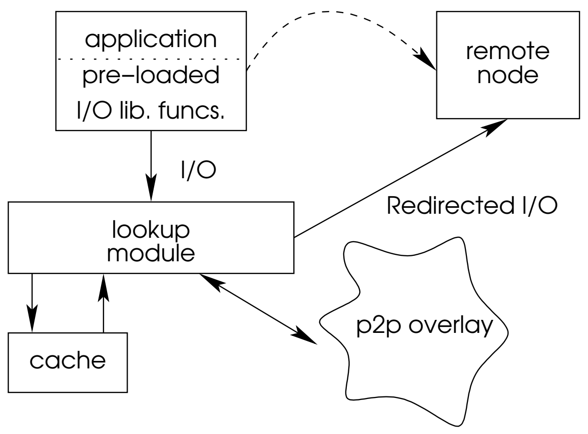

Figure 4. Interfacing PeerStripe with applications. The dotted line shows the I/O as perceived by the application.

we have not explicitly handled limiting chunk sizes based on such factors, our design allows for selecting chunk sizes according to local node policies, which can capture such factors.

5. Implementation

The system as discussed in the previous section was implemented with about 8000 lines of Java code using FreePastry [10] – the publicly available version of the Pastry API. We do not expect our Java-based implementation to become a performance bottleneck as the Java code is mainly used for locating remotely stored chunks where any processing delay due to use of Java is expected to be overshadowed by the network latency. Moreover, any actual transfer of data is done directly between nodes using standard techniques, and does not involve the p2p overlay.

To allow user programs to access the PeerStripe API without requiring any special changes to the source code or recompilation, we developed a library that interposes itself between the application and the standard libraries, and redirects the application's I/O as shown in Figure 4. The library consists of 259 lines of C code and utilizes standard techniques for redirecting library calls.

When an application uses our library, any `open` I/O call for files are sent to the *lookup* module that determines and locate the chunk that contains the portion of the file being accessed. Then, the *lookup* module determines the node storing the chunk using the p2p overlay as described in Section 4.1. For faster operation, the module also maintains a local cache of recently accessed chunks and the remote nodes on which the chunks are stored. Once the storing node is known, the I/O request is sent to it. Any subsequent `read` and `write` calls to the file are redirected to the remote node. Finally, the `close` call is also redirected to clear the state of the local file descriptor so it can be reused later. In this way, our implementation is able to transparently redirect I/Os from applications to distributed storage nodes.

6. Evaluation

In this section, we present a detailed evaluation of our system. Given the dynamic characteristics of our target environment, we rely on large-scale simulations to test PeerStripe in a controlled environment. However, later we present a case study and a verification of PeerStripe design on the PlanetLab [26] test bed using an actual implementation.

6.1. Simulations Results

Methodology We utilized the simulator mode of Pastry [30] to create a 10000-node directly connected network, where each simulated node runs an instance of our code. Moreover, to compare PeerStripe with others, we adapted CFS [9] and PAST [31] to run in our simulated environment.

We assigned the storage capacities of our simulated nodes following the recommendations of recent studies regarding available disk space on typical desktop grid nodes [14, 35]. Each simulated node was assigned a capacity based on a normal distribution with a mean and variance of 45 GB and 10 GB, respectively, resulting in a total simulated capacity of 439.1 TB.

To drive our simulations, we collected a file system trace from various video hosting websites, Linux mirror websites that serve distribution images, as well as from various departmental servers. Since our system is designed for large files, we filtered out all files smaller than 50 MB (based on large files used in works such as [24]). The resulting trace contained information for about 1.2 million files, with mean size of 243 MB and a standard deviation of 55 MB. The total storage size required to store all the files in the trace is 278.7 TB.

For the purpose of these simulations, the limit on consecutive zero-sized chunks in our system was set to 5. The replication factor in PAST and CFS was set to 1, and no error coding was used in our system. The authors of CFS used a fixed chunk size of 8 KB [9] in their evaluation, but given the large size of the files in our simulations we set the chunk size to 4 MB to reduce unnecessary DHT lookups. We considered a file insertion a success only if all the chunks of the files were successfully stored.

Finally, given the random `nodeId` assignment in our simulations, each case was simulated ten times; the results

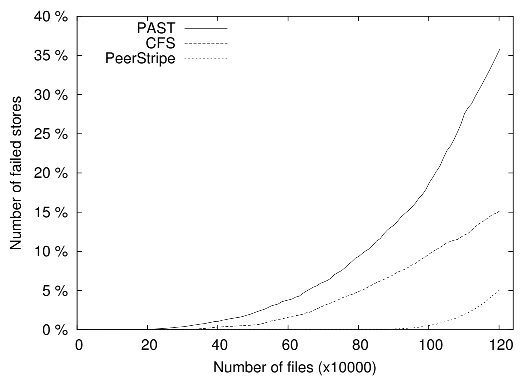

Figure 5. Ratio of failed file stores to number of files inserted.

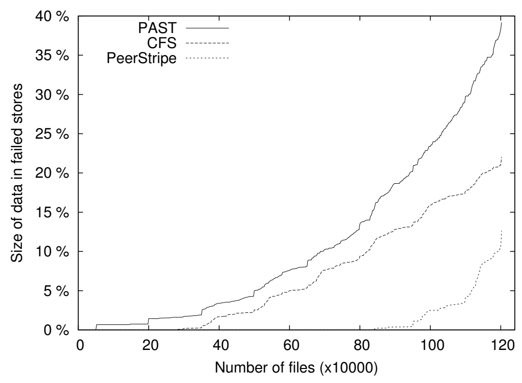

Figure 6. Ratio of failed data size to inserted data.

presented in the following sections represents the average (at each data point).

Results In the first set of experiments, we measured the number of successful file stores as files from the trace were inserted into the system. Figure 5 shows the results for the three cases of PAST, CFS, and PeerStripe. Initially, the storage is underutilized and the three schemes behave identically. However, as the storage utilization increases, the remaining space on many nodes become less than the size of the files being inserted. As a result, the number of failed stores in PAST starts to increase, and it fails to store 36.0% of the total files. This is of particular concern given that the total data to be inserted compared to the total available capacity, i.e., the expected utilization, is less than 64%. Similarly, CFS splits the files into blocks, and is therefore able to perform better than PAST by failing for 15.2% of the total files, however, this is still a large number of failures. The performance of CFS is expected to worsen further per our discussion of Section 3. Finally, PeerStripe is able to remedy the ill-effects of both PAST and CFS, and results in only 5.2% failures; an improvement by a factor of 7.0 and 2.9 compared to PAST and CFS, respectively.

Next, we measured the size of data that each of the three systems failed to store. Figure 6 shows the results. Here, we observe that PAST and CFS are unable to store as much as 39.2% and 22.0% of the data, respectively. In contrast our system was able to store almost all the data until about 800k files were inserted, only after that did it failed to store some files, with total amount of data that failed to be stored at 12.7%. This is an improvement by a factor of 3.1 and 1.7 compared to PAST and CFS, respectively.

Next, we determined the average number and size of chunks created under CFS and our system for these simulations. Since the size of a chunk is fixed at 4 MB in CFS, on average it results in the files being split into 61.25 chunks, with a standard deviation (sd) of 13.8. In contrast the average size of chunks created under PeerStripe is 81.28 MB (sd=19.9 MB), which results in creation of on average 3.72 (sd=3.1) chunks per file. The reduction in the number of chunks by a factor of 16.5 on average enables PeerStripe to avoid an unnecessarily large number of p2p look-ups and to provide performance similar to that of PAST but with the added capability to store large files.

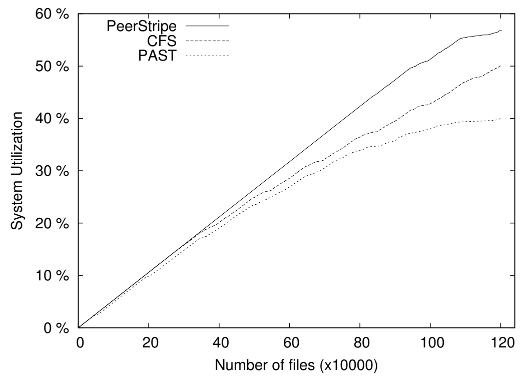

Figure 7. System storage utilization under the three scenarios.

In the next set of experiments, we determined the overall system capacity utilization under each of the three schemes studied. Figure 7 shows that all three schemes behave similarly in the beginning when the system is about 15% utilized. However, as more files are added, the utilization curves diverge. PAST and CFS are unable to store many of the files that are inserted as shown in earlier results, and as a result, achieve 16.8% and 6.7% less system utilization compared to PeerStripe, respectively. This shows that our system can utilize the available storage capacity more efficiently than the compared systems even at higher utilization.

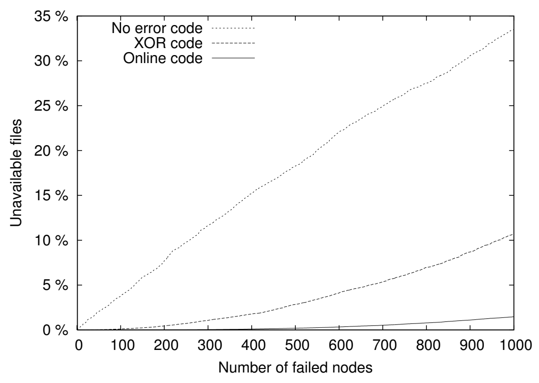

Figure 8. The number of unavailable files as nodes fail.

Erasure code	Encoded size		Encoding time	
	size (MB)	ovrhd.	time	ovrhd.
Null	4	0%	11	0%
XOR	6	50%	79	618 %
Online	4.12	3%	264	2300%

Table 1. Error coding overhead.

Nodes failed (percentage of total)	Data lost	Data regenerated		
	total (GB)	total (GB)	average (GB)	sd (GB)
10 percent	0	28044.35	28.04	78.95
20 percent	142.18	58625.78	29.31	80.02

Table 2. Effect of participant failure.

6.2. Fault Tolerance

File availability In order to determine the effectiveness of error coding in our system, we distributed the files from our trace to the 10000 simulated nodes, and counted the total number of available files in the system as 1000 randomly chosen nodes fail one-by-one. For this experiment we used a (2,3) XOR code, as well as an online code that could tolerate two simultaneous failures per chunk. We counted a file as available only if all the chunks of the file could be retrieved. We repeated this experiment for the cases of no error code, XOR code, and online code. Figure 8 shows the percentage of total files that became unavailable as nodes failed. The use of error coding resulted in 23% and 32% less failures for XOR code and online code, respectively, when 1000 nodes failed. The overall number of failures for online code was negligible (1.48%), and almost zero for up to 866 failed nodes. Moreover, these failures can be further reduced if encoded block re-creation is employed as described in Section 4.4.

Performance of error coding schemes We studied two erasure codes that can be used in our system, namely XOR and online code, and compared them against a NULL code that simply copies the input data to the output. For XOR code, we set the parameter $n = 2$ so that the number of blocks encoded per parity block is 2. The particular online code that we used follows the suggestions in [21], and has the tuning parameters of $q = 3$ and $\epsilon = .01$. For these runs, we used a chunk size of 4 MB, and used 4096 encoded blocks per chunk.

Table 1 shows the size of encoded blocks and time taken for encoding averaged over 10 runs. XOR encoding is a factor of 3.3 faster than that of the online code. However, although the online code is slower, the decoding can be started as soon as a block becomes available and can be overlapped with retrieval of other blocks. Moreover, online code has far less storage overhead as seen in the table, and therefore is a good candidate for use in PeerStripe.

Effects of participant churn In this experiment, we determined the effect of participant churn on PeerStripe. In particular, we studied the amount of data that is regenerated from other replicas/error-coded chunks as nodes leave the system due to failure. Upon failure of a node, its immediate neighbors spring into action. These neighbors identify the chunks of files which will now be mapped to them by the DHT, and start the recovery and chunk regeneration process. For this simulation, we failed up to 20% of the total participating nodes without any node recovery. After each node failure, we introduced a delay before the node's data is recovered on a neighboring node. This delay is proportional to the size of the data being recovered and serves to simulate the time it would take the data to be recovered in a real system. This delay also enables us to determine how the system would behave under multiple consecutive failures where data recovery due to a previous failure is not yet complete. For each failure, we logged the size of data that needs to be regenerated as well as the total size of data that has become unavailable.

Table 2 shows the results. We observed that for the traces used, an average of 29.3 GB of data was regenerated per failure after up to 20% of the nodes had failed, with a total of 58625.8 GB being regenerated. The experiment also showed that only 142.2 GB of data was lost even when 20% of the total nodes had failed. Finally, compared to the total data size of 278.7 TB, the data recreated per failure is quite small, i.e., 0.01%.

6.3. Multicast-Based Replica Management

This section evaluates the feasibility of using the Bullet [17] algorithm for disseminating replicas in our system. For this set of experiments we simulated how one source

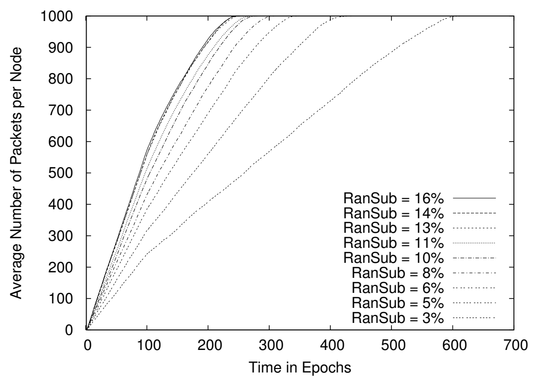

Figure 9. Average number of packets received per node with different RanSub values over a period of time.

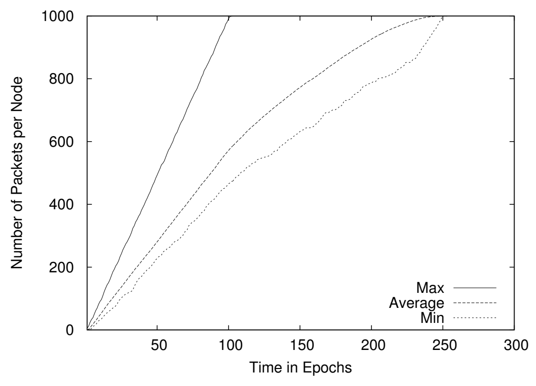

Figure 10. Average, minimum, and maximum number of packets received per node for a RanSub size of 16%.

File size (GB)	Time taken (s)		
	Whole file	CFS–Fixed size chunks (overhead)	PeerStripe (overhead)
1	151.0	169.0 (11.9 %)	176.4 (16.8 %)
2	277.1	330.8 (19.4 %)	302.4 (9.2 %)
4	529.1	654.6 (23.7 %)	554.5 (4.8 %)
8	1051.2	1320.0 (25.6 %)	1076.6 (2.4 %)
16	N/A	2637.0 (N/A)	2086.2 (N/A)
32	N/A	5243.9 (N/A)	4156.4 (N/A)
64	N/A	10441.8 (N/A)	8217.7 (N/A)
128	N/A	20881.5 (N/A)	16425.8 (N/A)

Table 3. Evaluation of PeerStripe using a simple Condor application.

node will distribute an encoded chunk to a number of replicas (32 in this simulation). We used a binary tree with a depth of five with the source node as the root of the tree and the recipient nodes at the leaf nodes. The setup included a total of 63 nodes. This simulation corresponds to an extreme case of creating 32 replicas, where in reality we expect the number of replicas to be small (about 3). For these experiments we divided a chunk into 1000 packets.

Our first experiment tested the replica creation time using different values of the RanSub [16] set size in the Bullet algorithm. Figure 9 shows the average number of packets received through the duration of the simulation for the values of RanSub set size ranging from 3% to 16% of the total nodes in the tree. It is observed that as the RanSub size is increased, its effect decreases, and begins to stabilize around 8 percent. This shows that the RanSub size only effects the distribution time up to a certain point and then the distribution time becomes independent. This gives us an idea of what RanSub value should be chosen for our system in real applications.

In the next experiment, we examined how evenly the tree is saturated with the packets, i.e., how evenly the Bullet algorithm distributes the replica packets. This experiment had the same setup as the previous experiment but with the RanSub value fixed at 16% of the total nodes in the tree. As Figure 10 shows, the distribution of the replica data is close to linear for the maximum, average, and minimum number of blocks per node. This shows the even distribution of data over time, and that the Bullet algorithm can indeed be used for effective replica creation in our system.

6.4. Case study: Interfacing with Condor

In this section, we discuss how we interfaced PeerStripe with Condor [19], a well-established grid environment that enables high throughput computing using off-the-shelf cost-effective components. For this case study, we used our library implementation of Section 5 to redirect I/O calls. Moreover, all participating nodes run Condor Version 6.4.7.

We created a simple Condor application, `bigCopy`, that in essence creates a copy of a specified file. We use this application to compare the working and performance of a CFS-based system that uses fixed chunks sizes, PeerStripe, and the original Condor. We utilized 32 laboratory machines at our department to set up a Condor pool connected using 100 MB/s Ethernet, where each node has an Intel Pentium 4 3.0 GHz processor, 1 GB RAM, 40 GB hard disk, and contributed storage space based on a uniform distribution between 2 GB and 15 GB, with mean and standard deviation of 10 GB and 3 GB, respectively. In this experiment, no error coding was employed, and enough retries were made for all three cases to store a chunk so as to ensure that all blocks can be stored.

Table 3 shows the results of this experiment, where each row corresponds to a run of `bigCopy` with increasing file sizes ranging from 1 GB to 128 GB. For each run, we started fresh by deleting all the files from the previous run, and creating a file with the stated size on a different machine than the 32 machines in the setup. Next, we ran `bigCopy` through Condor to create a copy of the file. The table shows whether the copying succeeded, and how long it took for the process.

As expected, we observe that both CFS and PeerStripe work for smaller file sizes, but the use of DHT introduces an overhead. There are two components of this overhead: a fixed component due to I/O redirection and code interposition, and a variable overhead due to p2p look-up operations to determine the locations of the chunks. While we expect the fixed overhead to be implementation dependent, the variable overhead is directly proportional to the number of chunks created, which is very large in CFS, but is dependent on node capacities in our system. For this experiment, since the entire file was accessed, the variable overhead grows with file size. However, this scenario presents the worst case, and typically only portions of a large file are accessed at a time, in which case the overhead is expected to be much less. Finally, as the file size is increased the advantage of our system becomes evident; it is able to find storage for the copy whereas the original scheme of storing on a single node fails due to unavailability of space. Moreover, note that as the file size increases the total time to run `bigCopy` is dominated by the transfer time. As a result, the relative overhead introduced by PeerStripe for transferring large files becomes very small (under 2.5% for a 8 GB file).

This experiment shows that PeerStripe is effective in storing large files with an acceptable overhead, and implies that it can be used in practical desktop grid scenarios, where the file sizes are larger than the capacity of individual participating nodes.

6.5. PlanetLab Verification

Next, we tested our implementation on the wide-area distributed testbed of PlanetLab [26]. For this purpose, we selected 40 different sites distributed across the country. Given the smaller storage capacity of PlanetLab nodes than our simulated nodes, we emulated each site to contribute storage space based on a scaled-down normal distribution with a mean and variance of 80 MB and 17 MB respectively. Next, we distributed files from a scaled-down version of our simulation trace, which contained 12,000 files with mean and standard deviation in size of 24 MB and 5 MB respectively.

We observed that compared to PAST and CFS, the number of failed stores when all files were inserted reduced by 330% and 105%, respectively. Moreover, PeerStripe achieved 63% system storage utilization, compared to the 52% and 47% of CFS and PAST, respectively. During the course of the experiment, 4 nodes failed. Without error coding, this would have resulted in the loss of about 10% of the stored data. However, our online codes were able to provide 98.6% availability through these failures.

While individual node lookup times were under a second, the actual data transfer time varied a lot. We believe that heavy use of PlanetLab during the time of our experiments contributed to these fluctuations. However, we were able to verify the feasibility of our design through these experiments.

6.6. Summary

Our evaluation has shown that PeerStripe can provide a reliable and robust distributed storage system for modern scientific applications. In particular, our simulations have shown that compared to PAST, for large files, PeerStripe reduced the number and size of file store failures by a factor of 7.0 and 3.1, respectively, and achieved 16.8% better overall system utilization. PeerStripe also reduced the number of chunks created compared to CFS by a factor of 16.5 on average, allowing fewer p2p look-ups and leading to performance similar to PAST. Our experiments with error coding showed that the fault tolerance and data availability needed for a desktop grid system can be achieved with our system through the use of error coding. The system also handles participant churn well with only 0.01% of data regenerated per failure for the traces used. We also examined the use of multicast for replica maintenance and found that this technique can be effectively used in PeerStripe. Moreover, our case study of interfacing PeerStripe as an I/O library with Condor proves that the system can be used in practical desktop grid scenarios with acceptable overhead. Finally, we have verified the design of PeerStripe over the PlanetLab testbed and shown that the proposed design behaves as expected.

7. Conclusion

In this paper, we have presented the design and evaluation of a contributory storage system, PeerStripe. PeerStripe uses p2p overlay networks to establish robust, scalable, and reliable distributed storage. It employs the techniques of striping and error coding to support transparent storage of very large data files across multiple distributed nodes, and exports a simple yet effective interface to users and applications. The detailed evaluation of our system has shown that it performs better than existing systems in a dynamic setting, can store files that are larger than the capacity of individual participants, is reliable, and responds well to participant churn. We have also proposed the use of multicast for replica maintenance and believe that such an approach can be used in target environments. The efficient and simple design of PeerStripe implies that it can be readily deployed and interfaced with different applications, and therefore can serve as a storage system for today's desktop grid environments.

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*, 2002.

[3] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc. NSDI*, 2005.

[4] A. Bassi, M. Beck, T. Moore, J. S. Plank, M. Swany, R. Wolski, and G. Fagg. The internet backplane protocol: A study in resource sharing. *Future Generation Computing Systems*, 19(4):551–561, 2003.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proc. USENIX NSDI*, 2004.

[6] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *JPDC*, 63(5):597–610, 2003.

[7] J. Cipar, M. D. Corner, and E. D. Berger. TFS: A transparent file system for contributory storage. In *Proc. USENIX FAST'07*, 2007.

[8] B. Cohen. BitTorrent Protocol Specification, May 2007. `http://www.bittorrent.org/protocol.html`.

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

[10] Druschel et. al. Freepastry, 2004. `http://freepastry.rice.edu/`.

[11] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. ACM SOSP*, 2003.

[13] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. 2nd USENIX NSDI*, 2005.

[14] H. Huang, J. Karpovich, and A. Grimshaw. A feasibility study of a virtual storage system for large organizations. In *Proc. 1st VTDC workshop*,2006.

[15] C. L. B. III. Utilizing large distributed computational resources in molecular biophysics. In *First Advanced Topics Workshop on Desktop Grids: Critical Systems and Applications Research*, 2003.

[16] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. M. Vahdat. Using random subsets to build scalable network services. In *Proc. 4th USENIX USITS*, 2003.

[17] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. ACM SOSP*, 2003.

[18] M. Livny. If you can do it on the Desktop you can do it everywhere. In *First Advanced Topics Workshop on Desktop Grids: Critical Systems and Applications Research*, 2003.

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

[20] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The lockss peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 23(1):2–50, 2005.

[21] P. Maymounkov. Online Codes. Technical Report TR2003-883, New York University, New York University, New York, Nov 2002.

[22] C. Miller, P. Butler, A. Shah, and A. R. Butt. PeerStripe: A P2P-based large-file storage for desktop grids. In *Proc. IEEE HPDC*, 2007.

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

[24] K. Park and V. S. Pai. Scale and performance in the coblitz large-file distribution service. In *Proc. USENIX NSDI*, 2006.

[25] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. ACM*, 1988.

[26] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First ACM Workshop on Hot Topics in Networks (HotNets-I)*, 2002.

[27] J. S. Plank. Erasure codes for storage applications. Tutorial Slides, presented at *4th Usenix FAST*, `http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html`, 2005.

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

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

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

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

[32] 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*, 2001.

[33] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Gathering at the well:Creating communities for Grid I/O. In *Proc. ACM/IEEE SC2001*, 2001.

[34] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5):757–768, 1999.

[35] S. Yang, A. R. Butt, X. Fang, Y. C. Hu, and S. P. Midkiff. A Fair, Secure and Trustworthy Peer-to-Peer Based Cycle-Sharing System. *Journal of Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):265–286, 2006.

[36] 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.