On Utilization of Contributory Storage in Desktop Grids

Chreston Miller

Patrick Butler

Ankur Shah

Ali R. Butt

May 1, 2007

Department of computer Science

Virginia Polytechnic Institute and State University

Blacksburg, VA 24061

Contents

1 Introduction **1**

2 Survey of Related work **3**

2.1 P2P-based storage . . . 3

2.2 Erasure codes . . . 4

2.3 Data transfer using multicast . . . 5

3 Motivation **6**

4 Design **7**

4.1 Overview . . . 7

4.2 Chunk storage and error coding . . . 8

4.3 Determining chunk sizes . . . 9

4.4 Fault tolerance and security . . . 10

4.4.1 Managing replicas . . . 10

4.5 Discussion . . . 11

5 Implementation **12**

6 Evaluation **13**

6.1 Simulations results . . . 13

6.2 Fault tolerance . . . 15

6.3 Multicast-based replica management . . . 16

6.4 Case study: Interfacing with Condor . . . 17

6.5 Summary . . . 18

7 Conclusion **19**

Abstract

The availability of desktop grids and shared computing platforms has popularized the use of contributory resources, such as desktops, 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 report, we propose a transparent distributed storage system that harnesses the storage contributed by 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) ingenious 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 simulations and an actual implementation show that our system can provide reliable and efficient storage with large file support for desktop grid applications.

1 Introduction

In recent years, the modern desktop has become a powerful resource that has the capability to support far more complex and demanding applications than typical desktop use. 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 [26] and Entropia PC Grids [11, 9] are being designed and deployed, their use as resource providers for modern scientific applications is becoming increasingly popular [25, 20].

While the focus of the desktop grids has mainly been on providing computational resources to execute user submitted jobs, e.g., Condor [26], addressing the ever-increasing storage demands of the 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 [26], or copying between the submission and execution machines, e.g., as in Globus [17], 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 [4]. The large size and dynamic nature of data used by modern grid applications [39] 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 [18, 13, 37, 35, 33, 23, 15, 28, 2, 36, 7] 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 report, we propose to develop a p2p storage system that provides an economical and efficient storage solution for large data files. Our goal is an elegant and simple system design [24] 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 has the features of scalability, self-organization, reliability, and composability for target environments of various sizes. A unique feature of our system is that instead of storing entire files on individual nodes, it splits the files into varying sized chunks and then stores these chunks separately on heterogeneous nodes distributed across a wide-area network. This approach is inspired by the data striping techniques employed in local-area RAID [30] clusters. As a result, unlike previously proposed approaches such as PAST [35], the size of a file that can be stored in our system 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 our system provides fault-tolerance and data availability despite churn of system participants.

Users and applications can access the distributed storage exported by our system by using its APIs that allow storing and retrieval of entire as well as portions of files, and our system can easily be interfaced with existing as well as new applications. The proposed system 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 report 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 support a rich set of features such as mobility and location transparency, self-organization, load-balancing, and decentralized operation;

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

3. An exploration of multicast techniques for data replication;

4. An implementation of the proposed system that allows easy integration of our system with applications; and

5. A detailed evaluation of the proposed system via large-scale simulations and an implementation study of how it can be interfaced with Condor [26].

The rest of the report 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 our system, and finally Section 7 concludes this report.

2 Survey of Related work

The design of our proposed system is based on the observation that typical desktop machines in academic and corporate settings have a large amount of unused disk space [19, 7]. 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 [35, 26, 7, 11, 9, 17, 6, 40, 14].

In the following sections, we summarize the related technologies that serve as building blocks for this work.

2.1 P2P-based storage

Structured p2p overlay networks such as CAN[32], Chord[38], Pastry[34], and Tapestry[41] 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 `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 can be used for transparent distribution of files on participating nodes in p2p-based storage systems.

The use of p2p techniques in providing large-scale, distributed storage for a variety of applications is explored by a number of works [18, 13, 37, 35, 33, 23, 15, 28, 2, 36, 7]. These systems provide strong persistence and reliability, and are complimentary to the design of this work. There has also been research done in providing applications transparent access to the p2p-based storage. Systems in this category include Kosha [7] that provides a Network File System interface to the p2p storage system, and TFS [12] that provides transparent access to contributory storage and aims to contribute maximum disk space with the least effect on the local file system in terms of performance and capacity. However, these systems require access/modifications to the host kernel and may not be suitable in a grid environment.

Our proposed system shares with the above mentioned 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 in the system. We do not aim to provide a general-purpose file system 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. Next, we discuss two p2p-storage systems that we have used in our evaluation in more detail.

PAST [35] is a large-scale, Internet-based, storage utility, which uses the p2p network provided by Pastry [34] 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 stores 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 file chunks, and adapts the core PAST functions to better handle large files.

CFS [15] 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 (8K) 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. The component that stores data is referred to as 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 [38] 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.

2.2 Erasure codes

A well-established technique for providing high availability and reliability in data storage systems is error coding. Since our proposed system can rely on the underlying network protocols and hardware to detect and correct simple errors such as *bit flips* or channel errors, the main task that our design faces is to recover lost data that was stored on a failed participant node. This problem can be addressed using erasure codes, which are error codes that support data recovery in the event of the loss of whole blocks of data.

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. The value of k depends on the kind of erasure code used. To quantify this overhead the parameter *rate* (r) is defined as the fraction $r = \frac{n}{n+k}$. The goal is to support recovery of the original data given a partial subset of the $(n + k)$ blocks [31]. The minimum size of this subset required for decoding is defined as $(1 + \epsilon)n$, where (ϵn) is a measure of how many extra blocks are required to decode n original blocks. There exists optimal erasure codes that support decoding of the original n blocks using only n encoded blocks, i.e. with $\epsilon = 0$. However, the calculation of such optimal codes is either CPU or memory intensive. To reduce this overhead, there exists sub-optimal erasure codes which allow for decoding the data using only $(1+\epsilon)n$ blocks for some $\epsilon > 0$. To summarize, r is an indicator of the number of extra blocks that will be created using an erasure code, while ϵ is an indicator of the number of encoded blocks required for decoding the original data.

The simplest erasure code is the parity check code, and is the erasure code used in RAID level 5 [30]. In parity check code, for every n input blocks to be encoded, an extra block that contains the XOR of the input blocks is added. A major drawback of this scheme is that it is very inefficient and can only tolerate the loss of one encoded block. For example, an $n = 2$ parity check code creates three encoded block for every two input blocks, which results in a space overhead of 50% (one extra block for two original blocks). Parity check does have the advantage of being fast due to its simple coding approach.

Recently, a new class of sub-optimal erasure codes, called rateless erasure codes [27, 31] have been proposed. The rateless codes 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 $(1+\epsilon)n$ blocks. There are several implementations of rateless erasure codes available, but the particular class of erasure codes we have studied in this work is the online code [27]. The online code uses two sub-optimal erasure codes, referred to as the outer and inner codes. The online code works by first applying the outer code to create a predetermined number (q) of auxiliary blocks. Each input block is then XORed with a pseudo-randomly chosen auxiliary block, and the XORed blocks are then processed by the inner code to yield the encoded blocks. Online code has the advantage of $O(1)$ encode time and $O(n)$ decode time per block.

In the context of our proposed 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 the 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.

The techniques of striping and error coding used in our system are the hallmark of RAID [30], which uses several storage devices in parallel to provide reliable storage for files. However, RAID is generally used in local storage devices and typically all the devices are similar and the rate of change in the RAID configuration by adding or removing devices is low. RAID is complementary to this work, and we adapt many of RAID's concepts in a wide-area distributed setting where nodes are heterogeneous and highly dynamic.

2.3 Data transfer using multicast

A number of systems such as Bullet [22], Shark [3], and CoBlitz [29] have explored the use of multicast and p2p-techniques for transferring large amounts of data between a source and a destination. Inspired by these systems, we have investigated data replication using multicast techniques of Bullet.

Bullet is a multicast system designed for efficiently distributing data on a network of nodes that are arranged in a logical tree. The root of the tree is the source of the data to be distributed, and the leaf nodes of the tree represent the final receivers of the data. Each node receives data from his parent based on a RanSub [21] which consists of the information regarding a subset of the total nodes and what data those nodes have received. Data is transferred in epochs that consist of a distribute and a collect phase. The distribute phase sends messages down the tree to each vertex until the leaves are reached. These messages consist of the RanSubs of the sending node, the parent of the sending node, and the Ransubs of the other children of the sending node. The collect phase sends messages up the tree once the distribute phase is complete. These messages compact each nodes RanSub into a smaller subset and send this new RanSub to its parent. This continues until the root is reached. In this way, each node has a subview of the whole tree and information regarding which nodes have what data. This information is useful since each node can use the information to intelligently distribute the data to where it is needed most. Then using this information, the root distributes the data recursively down the tree. Nodes in the tree can not only receive data from their parents but also from sibling nodes; one of the extra benefits of using Bullet. This is particularly useful when network bottlenecks further up in the tree cause slow transfer rates down certain paths of the tree. As a result of using RanSubs, each intermediate node in the tree has partially overlapping subsets of the data being distributed to the receiving leaf nodes. In this way, the system provides some overlap in the data residing at different nodes, which allows a node to be able to get the data it wants from multiple sources as needed. Therefore, the Bullet algorithm provides a suitable structure for our need to distribute replicas intelligently.

3 Motivation

We have discussed a number of p2p-based storage systems in Section 2.1. While these systems provide a number of features necessary for applying a p2p storage system in a desktop grid environment, we observed several shortcomings in these systems: maximum size of data files that can be stored in the system limited to storage capacity of individual contributors [35]; use of simple replication to k replicas, which only provides reliability against k simultaneous failures [35, 15, 7] and wastes storage space if k is set too large; supporting large files by dividing them in fixed size blocks [15], which results in scalability issues as the blocks per file increase directly with the size of the file. This work aims to address some of these challenges, in particular the handling of large data files.

Several systems such as CFS [15] store large data files using a shared pool of storage resources by dividing files into fixed size blocks. 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 [35], 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 the 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 look-up 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 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.

The goal of our work is to learn from lessons of these previously proposed systems, and introduce novel techniques such as multi-sized chunk striping to overcome large number of chunks per file, as well as to use error-coding for improved reliability.

4 Design

In this section, we present the design of our proposed system. It is assumed that a set of participating machines are available and willing to contribute storage space towards the system-wide shared storage. Moreover, the participants faithfully implement the underlying communication and the protocols of our system. Similar to PAST [35], our system also assumes that all files to be stored in the system have unique file names.

The design of our system allows users to store and retrieve entire files, as well as access portions of files. A portion is accessed by specifying its length and its offset with respect to the beginning of the file. In the following sections, we first give an overview of our system. Next, we give details of different aspects of our system. For this discussion, we refer to the machines that intend to participate or are participating in our system as "nodes".

4.1 Overview

The first step of our system is to establish a pool of shared storage resources. We accomplish this task by using the communication substrate provided by Pastry [34] to arrange the nodes in a p2p overlay network. Our use of structured p2p networks implies that the proposed system can support features such as fault-tolerance, resiliency, high-availability and self-organization of participants. Figure 1 illustrates a new node joining the overlay. Once nodes become part of the overlay, they can reach each other and utilize and contribute to the storage in the system. Hence, the desired pool of shared storage resources is established and is ready to be used.

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, the system splits a file into chunks, and stores 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 in the system, we utilize the 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

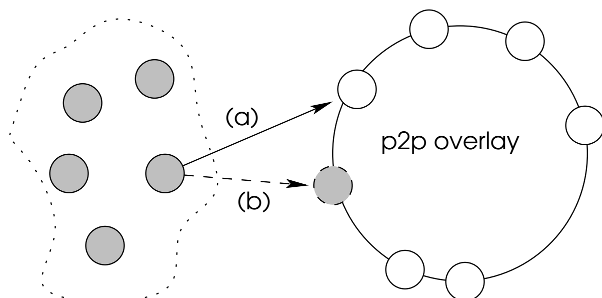

Figure 1. A node joining the participant overlay. (a) The new node (shown on the left) sends a message to one of the participating nodes (shown on the right). (b) The new node becomes part of the overlay and starts contributing storage.

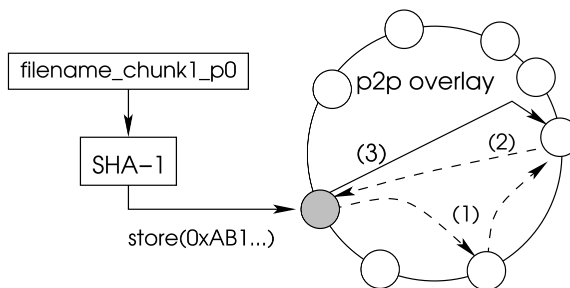

Figure 2. A chunk is stored in the system from the shaded node. (1) `lookUp` message. (2) Acknowledgment with IP address of the target node. (3) Actual store of the chunk (over IP network).

performing SHA-1 [1] hash on the chunk name. The UID is then used as a key to send out a `lookUp` message in the overlay. The 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 our system 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. An example of this process is shown in Figure 2.

4.2 Chunk storage and error coding

In this section, we discuss how our system stores chunks of a file. For this discussion we assume that the sizes of chunks of a file are known, in the next section we will present how these chunk sizes are determined.

Each chunk is named as *filename_ChunkNo*, e.g., *testImageFile_2* represents the second chunk of the file *testImageFile*. This naming convention is chosen as a means for determining the name of the file a chunk belongs to, and alleviates the need for maintaining mappings of chunks to files and vice versa. A drawback of this naming convention is that it complicates renaming of a file as all chunks belonging to the file should be renamed (and possibly moved based on new DHT-mapping). However, we argue that the targeted large files such as medical images are named based on their contents, and a rename is a rare operation.

To ensure file availability in the face of node failures, we employ error coding. We considered two options regarding the granularity at which the error coding should be applied. One is to perform error coding across chunks, i.e., n chunks are encoded into m chunks and stored in the system. The main issue with this option is that, in the case of a failure, recovering a chunk requires accessing at least n encoded chunks. The size of the n chunks is the size of the stored file, and given that we are dealing with very large files, such a recovery mechanism is very expensive in terms of both time and resources consumed. An alternative that we have opted for is to encode each chunk individually as described next.

A chunk to be encoded is passed to an error coding algorithm that divides the chunk into n equal size blocks, calculates erasure codes across the blocks, and generates m encoded blocks. The encoded blocks for the chunk X are named *filename_X_ECB*, where ECB is the error coded block number and ranges from 1 to m. The error coded blocks are stored in the system similar to the storing of chunks as described in the previous section. Since the names of the encoded blocks are different from each other, with a high probability they are stored on different nodes and thus are less prone to simultaneous failures. Due to the built-in redundancy of erasure codes, our system can retrieve the original chunk even if some of the m encoded blocks are lost due to failures.

A disadvantage of using varying size chunks is 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 offset values. Our system creates the chunk allocation table when a file is stored, and stores it in the p2p storage under the name *filename.CAT*. Figure 3 shows an example *CAT* file. To look up a file, the system locates the *filename.CAT* file, and uses its contents to retrieve all or some of the file's chunks as requested by the user.

In summary, the following sequence of events happen when a file is stored. The file is first split into chunks. Each chunk is then divided into n blocks and error coded to give m encoded blocks. Finally, the encoded blocks are stored

```
(1)   0,5242880
(2)   5242880,26083328
(3)   26083327,52297728
(4)   52297729,86114304
(5)   86114305,86114304
(6)   86114305,104856576
```

Figure 3. Example contents of a CAT file. Each line represents a chunk. There are six chunks in the file, and the total size of the file is about 100 MB. Chunk #5 is empty.

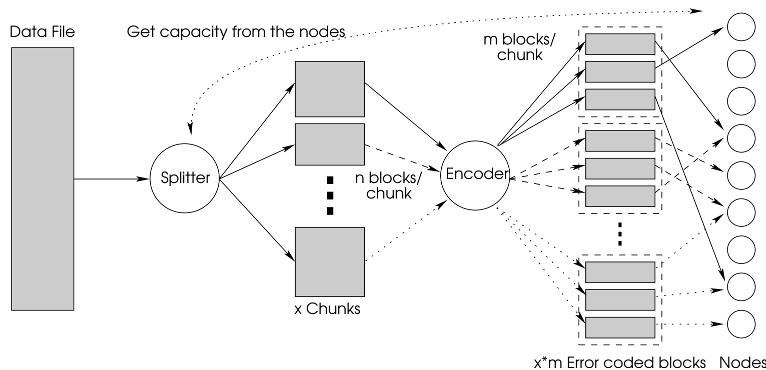

Figure 4. The various steps of storing a file in our system.

in the shared storage pool. The associated *CAT* file for the file is also created and stored in the system. Similarly, retrieval of an entire file or a portion of the file involves the following sequence of events. The system first retrieves the associated *CAT* file and determines the number of the chunk to retrieve and the name of the required encoded blocks using our naming convention. Next, enough blocks are retrieved to allow decoding of the chunk. The process is repeated until the desired number of chunks is retrieved. 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, our system will locate the encoded blocks: $myTestFile_x_y; 0 \leq x < 3, y$ any two in $\{0, 1, 2\}$.

4.3 Determining chunk sizes

In this section, we discuss how we determine the size of chunks for a file. First, we use the information about the currently used erasure codes to determine how many encoded blocks will be created per chunk. This information is static per erasure code, e.g., a simple (2, 3) XOR code creates three encoded blocks from every two input blocks. Next, we determine the names of the first set of encoded blocks belonging to the first chunk. This step is simple given our naming convention of *filename_chunkNo_ECB*. Note that only the names of the encoded blocks are created, and not the actual chunk or encoded blocks. Then these encoded block names are used to create message keys and send `getCapacity` messages on the p2p overlay. The messages are received at the nodes that will later store the encoded blocks. These nodes reply with the maximum size of an encoded block that they are willing to store. This size is determined by the remote nodes' local policies and can be zero, which indicates that a remote node is either out of space or unwilling to store data. The space is simply reported in the reply to `getCapacity` and is not reserved for the block. Upon receiving the replies, we determine the maximum block size that the remote nodes can store. Using the block size and the erasure code information, we can determine the maximum size of the first chunk. For example, if the maximum block size returned is 10 MB, under the above (2, 3) XOR code, the chunk size can be 20 MB. Next, the determined chunk size (or the size of the remaining portion of the file, whichever is less) is used to create a chunk of the file. The chunk is error coded and stored as discussed in the previous section. The process is repeated until all the data in the file is stored. Figure 4 shows this process in action.

There is a possibility that the available space on a remote node is used in the time between the reply to the `getCapacity` message and the actual store of the blocks. If this happens, the system can simply treat the current chunk as a chunk

of size zero, and continue normally. Also, to reduce the occurrence of this event, a node may choose to only report a fraction of its actual available capacity per `getCapacity` message. This allows the node to serve multiple stores simultaneously.

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 and security

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 it. For these blocks, the node then uses our naming convention to determine the associated chunk, and starts the process of re-creating the lost encoded block using the remaining encoded blocks. Note that the newly created encoded block may not be exactly the same as the one that has been lost, but it is functionally equal.

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 be avoided either 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; or by making a node consult its neighbors before replying to `getCapacity` messages and reserving space for potentially storing a neighbor's workload. We have adopted the former.

Another point of failure is the *CAT* file associated with each file. We employ simple replication of these files on neighbor nodes, and in case of failure of a node, create new replicas. This scheme of replication is similar to that used in PAST [35] and Kosha [7]. However, our system has the advantage that it can re-create a *CAT* file in case of loss. This is done by incrementally looking up chunks of a file and determining their size. In case a particular chunk is not found, it can either mean a zero-sized chunk or end of the file. Since, we limit the number of consecutive zero-sized chunks, we can always determine the exact end of the file by continuing searching for the next chunk up to one more than this limit. Re-creating a *CAT* file is a time-consuming process, but given the active replication, it will only be called upon in rare events.

4.4.1 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

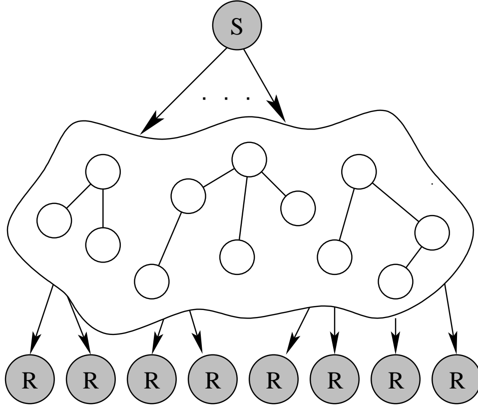

Figure 5. An example multicast tree structure for simultaneously creating replicas on nodes R of a chunk from source S.

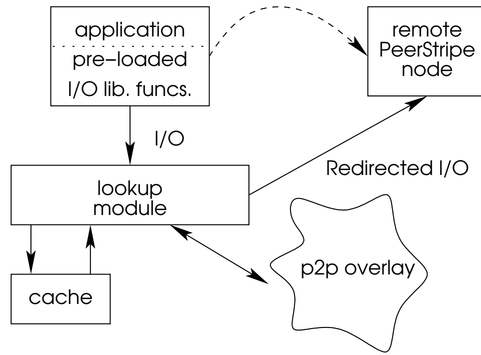

Figure 6. Interfacing with applications. The dotted line shows the I/O as perceived by the application.

in many systems [35, 7, 15], we utilized a multicast scheme to simultaneously create k replicas.

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 [22] 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 5.

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. Staring from the source node, we picked K closest nodes from the routing table as children, and then continued the steps as we moved towards the identifiers of the target node. As a result, the desired locality-aware tree was 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 simply use the Bullet algorithm to multicast the data to the k replicas.

Finally, wide-area file storage raises issues of trust and consistency. Our system can utilize the multitude of approaches that have been developed to address these problems, such as encryption [13], agreement protocols [10, 5], and logs [28]. A detail discussion of such approaches is out of the scope of this report.

4.5 Discussion

The design of our system results in dividing large files into relatively few chunks. However, a number of systems [3, 29] 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 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 bound for chunk sizes. While, 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. We continue to further investigate these trade-offs.

5 Implementation

In this section, we discuss our implementation of the proposed system. The design discussed in the previous section was implemented with about 8000 lines of Java code using FreePastry [16] – 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 location of 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 userspace programs to access the system's 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 6. The library consists of 259 lines of C code and utilizes standard techniques for redirecting library calls.

The interposition is achieved through the `LD_PRELOAD` environment variable available in Linux. Setting of this variable allows runtime loading of specific libraries prior to loading the standard libraries, which provides a means for overriding any standard library function. Using this feature, we overrode the `open`, `read`, `write`, and `close` functions in the GCC default libraries. The overriding functions use an RPC-based protocol to notify the local instance of our system when I/O requests are issued. The local instance then takes charge of performing the appropriate data location actions in the participant network.

The process starts when the application issues an `open` I/O call to a file. Instead of sending the I/O to the local machine, the interposed library passes it to a *lookup* module. The task of this module is to determine the chunk that contains the portion of the file being accessed, and locate the node in the shared storage pool which stores that chunk. For this purpose, the *lookup* module first consults a local cache to determine if it has recently handled an access to the chunk. The cache contains a file descriptor number and the remote node on which the file is stored. If the storing node of the chunk is already known, the I/O is redirected to that node and the process completes. If the storing node is not known, or the I/O to the chunk fails due to stale cache data (which is possible due to churn of participants), the *lookup* module determines the storing node using the p2p overlay as described in Section 4.1. Once the storing node is known, the system assigns a file descriptor for the file, and updates the cache with the nodes' information, and finally sends the I/O request to the node. For a `read` call, the system performs a `lookup` to determine the required block(s) and node(s) that contain the block(s) for the requested data. This information is then passed back to the library along with an error code and the library returns from `read` in the same manner as the original POSIX compliant `read` function is returned. The `write` function is handled in a similar manner. The overriding of the `close` function is done simply to clear the state of the 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, first, we evaluate our system through large-scale simulations to determine its capabilities in storing large files. Second, we investigate the fault tolerance of our system through measuring file availability, performance of error coding, and effects of participant churn. Third, we determine the feasibility of multicast-based replica maintenance. Finally, we present implementation results of using our system in an an example desktop grid system, Condor [26].

6.1 Simulations results

This section presents results from simulating a large number of nodes that implement the proposed design.

Methodology:

We utilized the simulator mode of Pastry [34] to create a 10000-node directly connected network, where each simulated node runs an instance of our code. This simulation approach has been used in a number of Pastry-based projects [34, 35, 8, 40]. Moreover, to compare our system with others, we adapted CFS [15] and PAST [35] 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 [19, 7]. 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 from various departmental servers. Since our system is designed for large files, we filtered all files smaller than 50 MB. Our choice of 50 MB as a minimum size was also based on large files used in works such as [29]. The resulting trace contained information for about 1.2 million files, with mean size of 243 MB and the standard deviation in size 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. For the case of our system, the nodes reported their entire capacity in response to the `getCapacity` messages. The authors of CFS have used a fixed chunk size of 8 KB [15] in their evaluation, but given the large size of the files we used in our simulations we set the chunk size to 4 MB to reduce unnecessary DHT look-ups. 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 presented in the following 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 7 shows the results for the three cases of PAST, CFS, and the proposed system. Initially, the system is underutilized and the three schemes behave identically. However, as the system 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

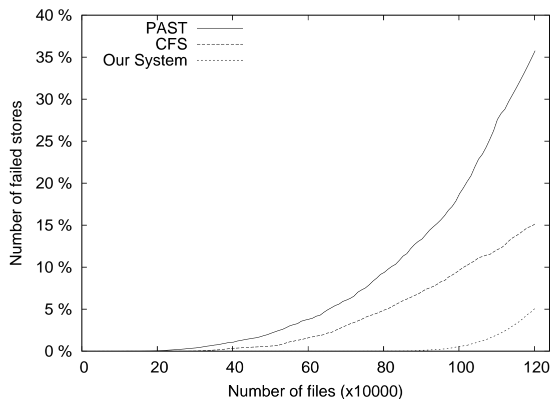

Figure 7. Total number of failed file stores as files are inserted, under the three scenarios, expressed as a percentage of total files inserted.

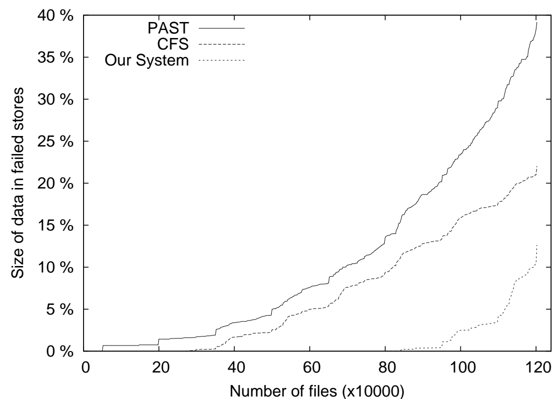

Figure 8. Total size of data that failed to be stored as files are inserted, under the three scenarios, expressed as a percentage of total size of the data inserted.

Scheme	Number of chunks		Size of chunks	
	avg	sd	avg	sd
CFS	61.25	13.8	4 MB	0
Our System	3.72	3.1	81.28 MB	19.9 MB

Table 1. Average and standard deviation of the number and size of chunks created under CFS and our system.

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, our system 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 8 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. The results are shown in Table 1. Since the size of a chunk is fixed in CFS, it results in the files being split into a large number of chunks, which on average is about a factor of 16.5 more than the number of chunks in our system. The reduced number of chunks enables our system 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.

In the next set of experiments, we determined the overall system capacity utilization under each of the three schemes studied. Figure 9 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, under-utilize the system by 30.4% and 10.7% compared to our system, respectively. This shows that our system can efficiently utilize the system storage capacity more efficiently than the compared systems even at higher utilization.

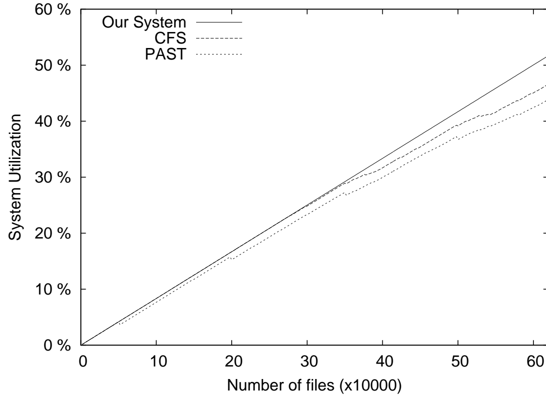

Figure 9. Overall system storage utilization as files are inserted, under the three scenarios, expressed as a percentage of total available storage in the system.

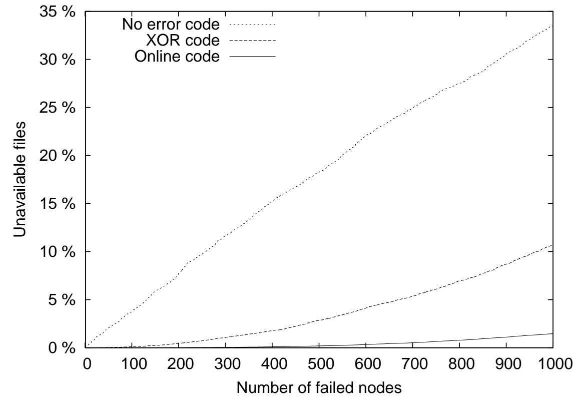

Figure 10. The number of unavailable files as nodes fail with and without error coding.

6.2 Fault tolerance

This section evaluates the fault tolerance characteristics of our system by studying the number of available files under node failures, and the performance overhead of error coding. We also study how participant churn will affect the system.

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 10 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. Hence, error coding is an effective means for ensuring fault tolerance in our system.

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 have used follows the suggestions in [27], 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 2 shows the size of encoded blocks and time taken for encoding and decoding averaged over 10 runs. XOR encoding and decoding is a factor of 3.3 and 19.7 faster than that of the online code, respectively. However, although the online code is slower, the decoding can be started as soon a block becomes available and can be overlapped with

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 2. Time consumed, and size of encoded data, in storing a 4 MB chunk. Overheads are with respect to the NULL code.

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 3. Data lost and regenerated after 10% and 20% nodes have failed. The average and standard deviation for regenerated data reported here is calculated for each failure.

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 our system.

Effects of participant churn:

In this experiment, we determined the effect of participant churn on our system. 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 3 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%. This shows that our system can handle participant churn well.

6.3 Multicast-based replica management

This section evaluates the feasibility of using the Bullet [22] algorithm for disseminating replicas in our system. For this test we simulated how one source node will distribute an encoded chunk to a number of replicas (32 in this simulation). We used a binary tree with a height of five with the source node as the root for 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 set size in the Bullet algorithm. Figure 11 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

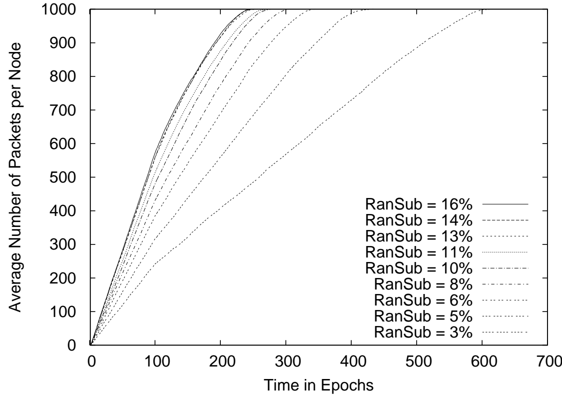

Figure 11. Average number of packets received per node with different values of RanSub over a period of time.

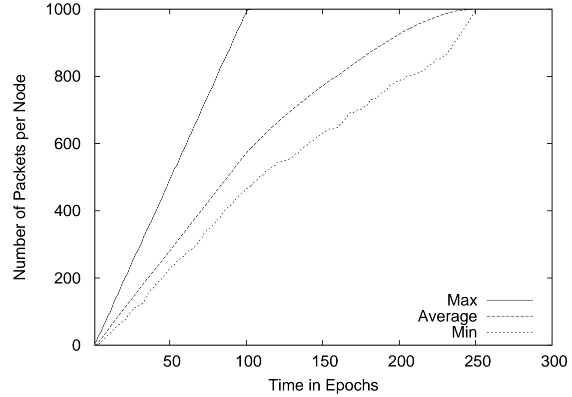

Figure 12. Average, minimum, and maximum number of packets received per node for a RanSub size of 16%.

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 12 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 our proposed system with Condor [26], a well-established cycle-sharing system 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, and we can start and stop client-side Condor I/O daemons on any of the machines to which I/O calls will be redirected.

For this study, 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-like system that uses fixed chunks sizes, our proposed system, and the original Condor. We utilized 32 laboratory machines at our department to set up a Condor pool connected using a 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 4 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

File size (GB)	Time taken (s)		
	Whole file	Fixed size chunks (overhead)	Varying size chunks (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 4. Comparison of the proposed storage system and fixed storage location scheme using a simple Condor application. The presented overhead is with respect to the whole file scheme.

the process.

As expected, we observe that both fixed and varying-size schemes 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 a system that uses fixed size chunks, 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 our system for transferring large files becomes very small (under 2.5% for a 8 GB file).

This experiment shows that our system 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 Summary

Our experiments have shown that our proposed system 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, our system reduced the number and size of file store failures by a factor of 7.0 and 3.1, respectively, and improved the overall system utilization by 30.4%. Our system also reduced the number of chunks created compared to CFS by a factor of 16.5 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 our system. Moreover, our case study of interfacing the system as an I/O library with Condor proves that the system can be used in practical desktop grid scenarios with acceptable overhead.

7 Conclusion

In this report, we have presented the design and evaluation of a contributory storage system. Our system 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. We evaluated our system through trace-driven 10000-node simulations and showed that it performs better than existing systems, achieves better than 99% file availability, and can store files that are larger than the capacity of individual participants. Additionally, the proposed system also responds well to participant churn with the amount of data regenerated per node failure being less than 0.01% of the total data in the traces used. These results indicate that the proposed system gives acceptable performance in a dynamic setting. We have also proposed the use of multicast for replica maintenance and believe that such an approach can be used in the target environments. The efficient and simple design of our approach 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*, December 2002.

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

[4] 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*, March 2004.

[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] A. R. Butt, X. Fang, Y. C. Hu, and S. Midkiff. Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing. In *Proc. 3rd Virtual Machine Research and Technology Symposium(VM'04)*, San Jose, CA, May 2004.

[7] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A Peer-to-Peer Enhancement for the Network File System. *Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):323–341, September 2006.

[8] A. R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of Condors. In *Proc. ACM/IEEE SC2003: International Conference for High Performance Computing and Communications*, Phoenix, AZ, November 2003.

[9] B. Calder, A. A. Chien, J. Wang, and D. Yang. The entropia virtual machine for desktop grids. In *Proc. 1st ACM/USENIX VEE*, June 2005.

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

[11] 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.

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

[13] 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).

[14] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

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

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

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

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

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

[20] 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*, November 2003.

[21] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. M. Vahdat. Using random subsets to build scalable network services. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, March 2003.

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

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

[24] B. Lampson. Hints for computer system design. *Proc. 9th ACM SOSP*, 17:33–48, 1983.

[25] 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*, November 2003.

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

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

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

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

[30] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

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

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

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

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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: International Conference for High Performance Computing and Communications*, Denver, CO, November 2001.

[40] 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. *Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):265–286, September 2006.

[41] 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.