Analysis of Erasure Coding in a Peer to Peer Backup System

George Nychis, Argyro Andreou, Deepti Chheda, and Alexander Giamas
Information Networking Institute
Carnegie Mellon University
{gnychis,aandreou,dchheda,agiamas}@andrew.cmu.edu

Abstract— **The peer to peer architecture model is widely used today for distributed systems deployment, which is increasingly becoming used as a model for disk back up systems. Currently, popular peer to peer back up systems use replication to ensure data redundancy, imposing a significant burden on system resources such as storage. Could the technique of erasure coding constitute a more suitable data redundancy scheme for such systems? In this paper we explore the prospect of deploying erasure coding in a peer to peer back up system and analyze how it performs against replication. Erasure coding and replication are compared in terms of CPU and network utilization, storage overhead, fault tolerance, end to end delay, and scalability. In addition to the performance analysis we also determine the turning point between erasure coding and replication with regard to peer availability. We provide suggestions for erasure coding parameters with respect to peer availability and file size based on this analysis.**

I. Introduction

The growing need of computational power, storage, and communication bandwidth in personal computing has led to decentralized peer to peer system implementations. These systems do not include dedicated servers for providing services, instead they include a number of personal computers operating in a cooperative manner to provide higher levels of quality of service, magnitudes greater computational power, disk space, and use of network resources.

The trend of peer to peer system deployment is also affecting disk backup systems. The primary goal of a disk backup system is to provide storage of data, ensuring that in the event of software failure, hardware failures, or even natural disaster, the data will still be recoverable. Traditional backup systems use centralized servers to provide mass storage for data that needs extra levels of redundancy, using the client-server architecture. However, backup system implementations are beginning to use the peer to peer architecture which can eliminate single points of failure to provide a greater level of fault tolerance, while increasing storage capacity and driving down costs.

Peer to peer disk backup systems can use replication to handle events of peer failures. Each backup in this scenario is replicated across the peers ensuring that even if only one peer is connected, the file will still be retrievable. Even though this failure handling scheme provides a high probability of retrieving files under severe peer failure conditions, it imposes large storage and network overhead. Current research has provided alternative failure handling methods that balance storage and network overhead with a higher level of fault tolerance under peer failure conditions.

In this paper we study the use of erasure coding as an alternative failure handling scheme to replication. Our aim is to determine whether the former would constitute a more balanced solution in terms of file retrieval capability under peer failure condition, network bandwidth, and storage overhead than the latter. To achieve this we began our implementation using HiSpread, an open source peer to peer backup system, and incorporated the PASIS[9] project's erasure coding library provided by the Parallel Data Lab at Carnegie Mellon University. For performance analysis of our experiments we used the Emulab[7] testbed located at the University of Utah.

The paper is organized as follows. In Section 2 we provide introductory information concerning erasure coding along with an example so that the concept is fully understood by the reader. In Section 3 we discuss related research to our topic. In Section 4 we provide a thorough description of how we incorporated the PASIS erasure coding library in HiSpread, referencing all of the challenges and problems we faced. In Section 5 we describe our experimental setup on Emulab, how we approached the issue what experiments should be performed, and how they should be performed. In Section 6 we provide a thorough analysis of the data we collected during our experiments concerning CPU, disk, throughput, as well as other metrics, with comparisons between erasure coding and replication. In Section 7 we provide insight on our future goals regarding this project, and lastly in Section 8 we conclude the paper.

II. Erasure Coding

Erasure codes are characterized by their ability to provide redundancy, and hence failure resistance, without the overhead of strict replication. Erasure coding divides a block into m fragments and then produce n redundant fragments, where $n > m$. A key property of erasure coding is that for reconstruction of the block, only m fragments are needed out of the n total fragments that resulted from the erasure coding procedure. This constitutes an erasure coding scheme of *m of n*.

For example, the erasure coding scheme *5 of 8* would mean that a block is first divided into 5 fragments and from these blocks, an additional 2 fragments (n - m) are created. Any 5 out of the 8 fragments that resulted out of the erasure coding procedure would be sufficient to reconstruct the original block.

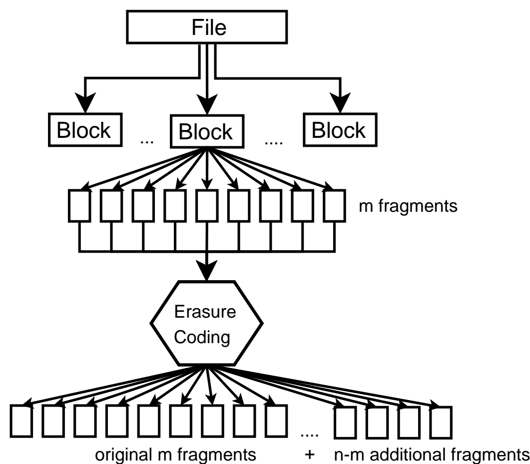

Fig. 1. Erasure coding example

Figure 1 depicts how erasure coding is applied to a file to produce *n* total fragments for a given block.

In our peer to peer backup system using erasure coding, we divide the file to be backed up into a series of blocks. Then we apply erasure coding on each of these blocks to produce the additional redundant fragments per block. For instance, if we are applying the erasure coding scheme *3 of 5*, each of the file blocks are divided into 3 data fragments (*m*). These data fragments are then used to produce 2 additional redundant fragments (*n-m*), for a total of 5 fragments (*n*). To recover each of the blocks we only need to have any 3 out of the 5 fragments.

It can be concluded from the above information that replication is a corner case of erasure coding where m is always equal to 1. RAID systems are also a case of erasure codes. For instance RAID levels 1 and 4 can be described by 1/2 and 4/5 erasure code schemes respectively. More detailed information about erasure coding can be found in [2].

III. Related Work

Comparing how erasure coding can provide a more robust solution in terms of efficient usage of network resources in file systems than replication is a topic that has not been researched sufficiently. The most notable efforts to quantify and compare erasure coding and replication in file systems have been deployed by Rodrigues and Liscov[1], Weatherspoon and Kubiatowicz[2] and Lin et al[3]. To implement our peer to peer backup system we used an open source peer to peer backup system, HiSpread, as well as the PASIS library for the erasure coding implementation. We incorporated the PASIS erasure coding library into HiSpread, changing a large portion of the initial system. This resulted to a peer to peer backup system that had the ability to back up files using either replication or erasure coding. The following are related works to our project.

A. Erasure Coding vs Replication for High Availability in DHTs

Rodrigues and Liscov[1] focus on comparing erasure coding and replication concerning peer to peer DHTs taking into account characteristics of the nodes including the overlay network. The comparison of the two redundancy schemes is performed in terms of peer availability, needed redundancy in relation with peer availability and bandwidth usage. The particular study concluded that in environments where peer availability is high, replication is preferred, and environments where peer availability is low, erasure coding is the preferred redundancy scheme. Experiments were conducted using PlanetLab, Overnet, and Farsite.

B. Erasure Coding vs Replication: A Quantitative Comparison

Weatherspoon and Kubiatowicz[2] quantitatively compare erasure coding and replication as potential redundancy schemes in distributed storage systems. The comparison of the two redundancy schemes is performed in terms of bandwidth usage and storage overhead using distributed storage systems with similar MTTF. In order to conduct the particular research the assumption that failures are independent and identically distributed had been adopted. Future work by the authors intends to explore the aspect of how replication and erasure coding perform under dependent and randomly distributed failures. The particular study concluded that erasure coding uses an order of magnitude less bandwidth and storage than replication.

C. Erasure Code Replication Revisited

Lin et al[3] focus on comparing erasure coding and replication with the goal to determine under which circumstances erasure coding is preferred over replication and vice versa. To achieve this a series of experiments is conducted measuring how erasure coding and replication respond to changes in peer availability, as well as the storage overhead each of these redundancy schemes imposes. In addition to these experiments, the authors conduct a thorough analytical derivation of their experimental results supporting their measurements and conclusions with mathematical analysis. The particular research resulted into two important conclusions: erasure coding should be preferred when the availability of the nodes is low and hence fault tolerance is required at the smallest possible cost against system resources and utilization. In contrast, replication should be preferred when node availability is high. The same results where derived by the Weatherspoon and Kubiatowicz[2] research study.

D. PASIS

The performance of survivable storage systems depends highly on the data distribution scheme selected. A data distribution algorithm includes a specific algorithm for data encoding and partitioning, as well as a set of values for its parameters. There are several data distribution algorithms including encryption, replication, erasure coding, and secret sharing. The PASIS project enables a better approach in selecting a suitable data distribution method. This includes three steps: enumerating possible data distribution schemes, modeling the consequences of each scheme and identifying the best data distribution scheme for a given set of security and availability parameters.

E. HiSpread

HiSpread is an open source peer to peer backup system whose redundancy scheme is replication. The degree of replication is defined by the user, and each file replica is stored on different peers participating in the particular group. The backup files are stored on the peers, taking the peers in sequential order, without taking into account properties of the peer system. The only two properties taken into account are peer availability and disk availability. HiSpread uses encryption to ensure that files are not corrupted during their transfer over the network though a hashed based signature scheme, which also ensures that the file was not altered during storage on the peer by recomputing it when the data is returned.

F. OceanStore

Kubiatowicz et al[4] propose an infrastructure for providing continuous access to persistent data spanning throughout the globe, using erasure coding and in particular Read-Solomon and Tornado codes. Erasure coding ensures reasonable storage overhead for the servers as well as greater reliability when retrieving the file. The authors opt in using erasure coding scheme with large m and n in an effort to achieve what they call "deep archival storage." The concept of deep archival storage is based on the notion that if m and n are large, it is very unlikely that enough servers will be down and the file will be unrecoverable.

IV. P2PeuR Implementation

The implementation of our peer to peer backup system, P2PeuR, was built on top of an already existing open source peer to peer project, HiSpread, and uses the erasure coding library from the PASIS project. By combining the efforts of the two projects, with major modifications to HiSpread and it's protocols, we were able to build a working peer to peer backup system with erasure coding. We will introduce the basics of the HiSpread implementation, our integration of the PASIS erasure coding libraries, and our modifications to HiSpread needed to fully implement our erasure coding peer to peer backup system.

A. HiSpread

HiSpread's implementation consists of several key components: a consistency protocol, versioning support, fault tolerance through replication, threading to handle concurrent connections, and a message passing protocol which is used during back up, retrieval, peer discovery, and block locating. HiSpread was implemented using standard C++, Borland specific API, and the Win32 API for its graphical interface.

We obtained the HiSpread source code from their SourceForge project and obtained a Borland C++Builder evaluation license to work on the project. Our intuition was that HiSpread would compile from the source code provided on their SourceForge project site, however the code did not include or mention the additional dependencies needed to build the project. The source code used the component TZipBuilder, which is a port of the Delphi TZipBuilder component to Borland C++Builder.

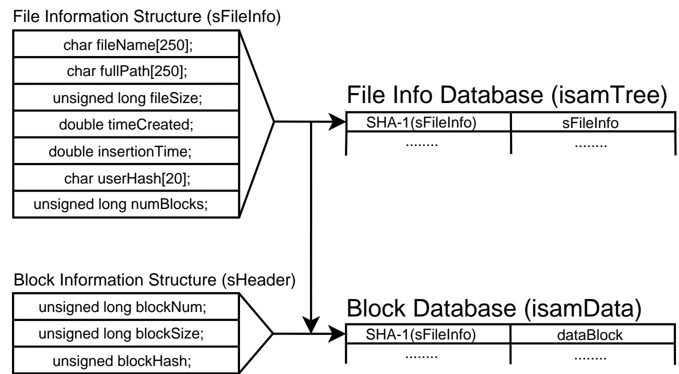

Fig. 2. HiSpread's database along with the file information and block information structures.

After finding this component and installing it into C++Builder, we were able to successfully build the project.

Before we talk about HiSpread's consistency protocol, it is important to understand how HiSpread stores and locates data blocks in its database. Whenever a peer chooses to back up a file, a file information structure (sFileInfo), is associated with it and each of its blocks that will be used to store the file in peer databases. This same structure is used to locate blocks for reconstruction. The file information structure includes the filename, full path to the file, the size of the data, the date the file was modified, the date the file was backed up, the hash of the peer's username, and the total number of blocks.

HiSpread uses two Indexed Sequential Access Method (ISAM) databases to manage indexes and data blocks for retrieval. The first database, labeled isamTree, stores the file information structures, indexed by a 20 byte SHA-1 hash of the sFileInfo structure. The second database, labeled isamData, stores the data blocks indexed by the SHA-1 hash of the sFileInfo structure the data block belongs to. Therefore, all data blocks that belong to the same file will have the same key and a sequential search will be used to retrieve the data blocks. The data at each index in the isamData database includes the block information structure (sHeader) and the data of the block. The sHeader structure consists of the block number, the size of the block, and a SHA-1 hash of the data block. The HiSpread databases and structures are visualized in figure 2.

When the user requests to back up a file, HiSpread first checks to see if versioning support is enabled. If versioning support is not enabled, HiSpread will search its local list of files backed up by the current user and see if any files match the filename and path. If no match is found, HiSpread will then prompt the user if they would like to overwrite the file. If the user selects yes then HiSpread will send out a message requesting the deletion of the old version of the file to all the peers storing blocks from it. The new blocks are then sent out with a new sFileInfo.

When versioning support is not enabled and a user requests to back up a file, the sFileInfo is created and the blocks are sent out. There is no need to anything extra since the insertionTime is included in the sFileInfo structure which would cause the SHA-1 hash to be different for the file, even if the user has already backed it up and no changes have been made. We

believe that this is poor versioning support and would allow the user to use unnecessary space on its peers by repeatedly backing up the file even if it has not changed.

As a peer receives a block to be stored in its database, it computers an SHA-1 hash of the sFileInfo that was sent with the block and checks its isamTree to determine if it already has any blocks for this file. If the hash exists in the isamTree then the blocks will simply be associated with that hash. If the hash does not exist then it will be created and all blocks will be associated with it.

This is very important to understanding the consistency protocol of HiSpread. The consistency protocol, based on the way blocks and file information structures are stored, would allow a user to login twice on two separate computers and back up the same file at the exact same time and not cause a crash or data blocks to be overwritten. However the two clients would be sending the same data and since blocks are not indexed by their hashes in the isamData database, there would have two instances of each block. This would require double the amount of storage than if the isamData database was also indexed with the hash of the block. This would have prevented storing the data twice.

Taking this scenario to the next step, when the user or users request the file back for reconstruction, both copies of each block will be sent back because HiSpread will sequentially search its isamData database and send every block that matches the sFileInfo hash. The negative aspect of this is that twice the amount of data will be received by the client that issued the request. The positive aspect is that if one of the peers crashed while trying to back up the file, it will not effect or corrupt the other peers back up of the same file. Therefore when the blocks are requested and returned, the file can still be reconstructed given at least one good copy of each block exists since the bad blocks will fail the SHA-1 hash computed when they are received back to the peer reconstructing the file.

The HiSpread database and the consistency protocol, given the information we have provided, is not optimal for disk space or performance. Given more time on the project, we would have opted to index blocks by their sFileInfo and blockHash to prevent the need for extra disk space and extra bandwidth to send/receive each block twice. This would have also needed changes to the consistency protocol to ensure that a crashing peer, when two peers were trying to backup the same file at the same time, did not corrupt already existing good blocks. For the time given for our project, we chose to leave the database and consistency protocol the same.

B. PASIS Erasure Coding Libraries

The erasure coding library integrated into P2PeuR was taken from the PASIS project at Carnegie Mellon Univerity's Parallel Data Lab. The library was written in the C++ programming language and implements Lagrange polynomial interpolation in a Galois Field. Erasure coding through Lagrange's polynomial interpolation will generate additional redundant points, given m points on a polynomial. The number of redundant points in m of n erasure coding is defined to be m - n. These additional redundant points generated will determine the level of fault tolerance. In m of n erasure coding, only m fragments are needed from n total fragments for reconstruction.

An interpolate function is provided by the library which is closest to Shamir's secret sharing algorithm[5], but also has properties of Rabin's information dispersal algorithm[6]. The library did not provide a functionality for splitting a file into blocks, computing block size, computing share size, or splitting the blocks into fragments. All of this was done in our P2PeuR application. To use the interpolate function we split a given file into blocks based on the block size which is computed by $M * share_size$. The share size is determined to be the size of the fragments to be sent out to the peers. After splitting the file into blocks and then further into fragments, the interpolate function is given m data fragments and n-m empty buffers which it will store its computed parity shares.

Although the use of the interpolation function was very straight forward, it's integration into the HiSpread project was everything but. HiSpread was coded in the C++ programming language using the Borland IDE, C++Builder. The integration of the HiSpread project and the erasure coding library should have been as easy as including the library's code into the HiSpread project. This simple task became difficult since Borland's compiler generated an internal compiler error trying to compile the erasure coding library. Several attempts were made to contact Borland for support with the broken compiler but none of our inquiries were returned. A method of integrating the library into HiSpread was needed because the HiSpread code was written mainly based on Borland specific libraries and data types. To port all of this Borland specific functionality to standard C++ such that we could have compiled it with g++, would have left us no time for implementation and analysis.

To successfully integrate the erasure coding library into the HiSpread project we compiled the library in g++ under Cygwin and created a Dynamic Linked Library. Creating the DLL was possible due to the fact that the interpolate function we needed from the PASIS library was written in C, and could therefore be compiled with an *extern "C";* wrapper around the function to prevent C++ name mangling. Module-Definition files were then created and dlltool was used to create the DLL.

C. P2PeuR

HiSpread was able to provide us with a simple consistency protocol, a message passing protocol, and a GUI so that our main focus could be on the integration of erasure coding and analysis. When trying to integrate erasure coding into HiSpread we quickly found that much of the message passing protocol, data structures, thread control, file dispersion and retrieval methods, as well as other implementation details all needed to be changed.

1) Additional Level of Data Decomposition and Reconstruction: The first major change in our conversion from HiSpread to P2PeuR, after integrating the erasure coding library, was to change the unit of data sent out to the peers during back up and retrieval from blocks to fragments. As shown in figure 2, the unit of data sent out in the original HiSpread implementation is a block, however as shown in figure 1, the unit of data needed for erasure coding is a

fragment. Therefore we needed to modify the decomposition and reconstruction methods for files to go from file to block to fragment, and visa versa.

To add the additional level of data decomposition and reconstruction we were able to keep the existing databases and structures in tact with the addition of a parityNum field, to what used to be the block header (sHeader) but is now considered the fragment header. Instead of blocks being stored in the isamData databases, fragments are now stored with a fragment header. Keeping track of which peers have which fragments was done by sending out fragments in a round robin fashion and only keeping the order of the peers. When fragments were retrieved we could compute which peers had which fragments by knowing this order.

2) Changes in Reconstruction Stage of Fragments: The major changes in these methods came in the reconstruction phase since all incoming data can no longer be immediately decrypted and written once it is received. The reason for this is that P2PeuR has two types of data, data blocks and parity. Since encryption is done at the block level, decryption could not be done until m of the n fragments for a block are retrieved. Furthermore, we need to detect if erasure coding is needed to reconstruct the file from parity before encryption can be done. This increased the complexity of the retrieval process.

A comparison between the original HiSpread data retrieval method and our P2PeuR retrieval method is shown in figure 3. To store parity as we received it, we created a temporary parity file on the filesystem. To temporarily store fragments we wrote the fragments directly to the final output file, regardless that they were encrypted since their encrypted length is no greater than their decrypted length except for the final fragment which we truncate. We stored these fragments in the final output file according to their proper block offsets as if the data were decrypted and ready. This gave us a place to temporarily store fragments and read them later for reconstruction and decryption.

3) Changes to Support Variable M, N, Share Size, and Block Size: These changes were made and tested using fixed M, N, share size, and block size through #defines and static character arrays since the original HiSpread code used all statically defined arrays. Once we successfully had a working erasure coding implementation with static erasure coding parameters the next step was to make all parameters variable. The reason for this is that all peers in the same network would need to have the same parameters to correctly receive and interpret fragments.

For instance, the original HiSpread implementation used a fixed block size and its message passing protocol used recv() socket calls expecting *block_size* bytes. The problem with this is that the receiving peer would need to have the same exact block size as all peers in the network to receive and interpret the blocks correctly. HiSpread does not include a block size in its block header that is sent with each block, nor does it include the block size in the sFileInfo structure which is also sent with each block. Therefore there was no way for HiSpread to dynamically determine the size of an incoming block.

To support variable erasure coding parameters the first step we took was allowing different sizes for the unit of data being

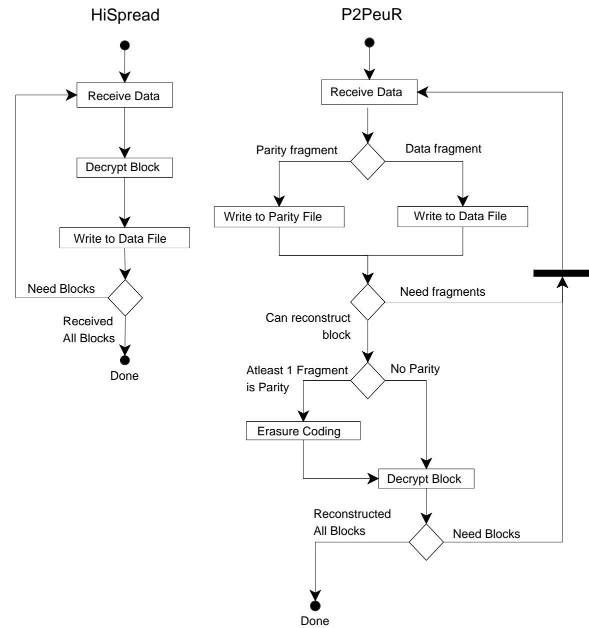

Fig. 3. Data retrieval and reconstruction comparison between HiSpread and P2PeuR

sent and received. In HiSpread this unit of data was blocks, but in our P2PeuR application it is fragments whose size are determined by share size. The first thing we chose to do was add m, n, *share_size*, and *block_size* to the sFileInfo structure. Our reasoning for this was that a main purpose of a back up system is to allow recovery after failure. If we did not include these erasure coding parameters in the sFileInfo structure, if a peer crashed and recovered without any information about files it had backed up, then the peer would not know what parameters were used to reconstruct the file after requesting it from its peers. Therefore adding these parameters would be essential to allow full recovery from failure not having any knowledge of backed up files.

4) Support for Backing Up Fragments of Dynamic Size: Fixing the problem of not knowing the fragment size when receiving a fragment to back up in the isamData database was simple after adding the parameters to the sFileInfo structure since it is sent with every block. When a peer received a back up fragment request it only had to check the sFileInfo structure included with each block to determine the amount of data it needed to receive.

5) Support for Restoring Fragments of Dynamic Size: Reading fragments from the database to send to a peer for reconstruction was not as intuitive. The reason for this is that the original HiSpread implementation sends only a hash of the sFileInfo when requesting data from a peer. Therefore once the peer receives this request which consists only of a hash, it can find the fragments in the database but not know how much data needs to be read. There were several methods we considered to support this functionality.

The simplest way to support this would be to add the share size in the fragment header. We opted to not use this method since it would greatly increase overhead per fragment for small share sizes or a large number of fragments. Another alternative

would be to do a lookup on the hash sent with the request in the isamTree for the corresponding file header. The file header could then be used to extract the share size from it. Looking up the file header from the database for every fragment did not seem optimal since since it would add computational overhead per request, increasing as the database increased in size since lookups are sequential. The scalability of the system would therefore decrease using this method.

The method we chose was for the requesting peer to add the share size to the request for fragments. This is possible even for a peer that has completely failed because a peer can send out a request for all sFileInfo's to its network of peers. The request will trigger all the peers to respond with all the sFileInfo's they have stored for the requesting peer's username and password. Whether the peer has failed and recovered or not, a peer always requests a file with a hash of the sFileInfo for the file it wants to reconstruct. Since the peer must have the sFileInfo to compute the hash for the request, it must have the share size used during the original backup. For these reasons we have determined adding the share size with the request the simplest method with the least amount of overhead and computations. The peer that receives the request will then assume all fragments in its database for the given hash have the share size sent with the request. This method leaves the system open for attacks by peers sending incorrect share sizes with requests, however our focus of our implementation is not security.

6) Statically Defined Buffers and Operations: Many other changes needed to be made to support variable erasure coding parameters since the original HiSpread implementation used statically defined buffers for temporarily storing data, receiving or sending data, and interpreting messages being passed. As an example, in the original HiSpread code the sBlock structure, used for storing and interpreting blocks, had two elements, an instance of sHeader and a statically allocated buffer to store the actual block data in. Therefore when sending/receiving blocks or reading/writing blocks to the database, a single memcpy() was used, reading or writing sizeof(sHeader+block_size) data, starting from the memory location structures memory location. Supporting variable block size, or fragment size in our case, required an instance of sHeader and a pointer which would point to dynamically allocated memory which contains the block data. Therefore a single memcpy() call could no longer be used and we had to ensure all sending/receiving fragments or reading/writing fragments to the database did not use a single memcpy() since it would overwrite the pointer after the sHeader and all data after the pointer, causing a buffer overflow. Along with this required properly allocating and freeing memory for all instances of our fragment structure.

We will not go into detail of all the changes that needed to be made to support the variable parameters, however it took a significant amount of time and debugging, but was essential to our system. Without these changes the system would be less distributed in manner since all peers would need the same parameters to back up and restore on each other.

7) Changing Thread Control for Performance: Another system change in the HiSpread code for our conversion to P2PeuR that we will talk about is the threading changes. When running initial benchmarks on our system we noticed dramatic increases in the time to backup files as the number of fragments increased and share size decreased. We spent a lot of time debugging our previous changes looking for something that could decrease the performance dramatically as the share size decreased. Not being able to find any errors in our changes, we looked to the original HiSpread code for the reason of our horrible performance. We noticed that whenever a fragment was to be sent, a TCP connection was setup to the peer, the fragment was transfered, and then the receiving peer tore down the connection after a single fragment was transfered, forcing the client to reconnect to send another fragment.

Since a TCP header is 20 bytes at minimum, the initial three way handshake plus the FIN and FIN-ACK incurs 10% overhead with a 1K share size. Worse than the overhead is the delay of setting up the connection and tearing it down for every fragment. We found this unacceptable and found it to have an adverse effect on our performance. To change this we setup a connection to all peers we were going to send a fragment to, before sending any fragments, and then tore down the connections when we finished sending all fragments. Doing this required changing the thread control on the receiver side though. HiSpread and P2PeuR use threading to accept and maintain multiple socket connections. Whenever an incoming connection is received, a thread is created, the command is processed, and the thread is closed which also tears down the socket connection. Therefore when a message to store data is received, the thread is created, it stores the data into the database, and then the thread is destroyed also closing the socket. To prevent this we modified the thread control for the data storage message. Instead of closing the thread after the data is stored, we changed the return value to not trigger the destruction of the thread and had the thread wait to receive more data. We then added our own message in the protocol which was sent to the peers at the end of the storage routing to close the socket connections and destroy the threads.

8) Additional Functionality to the GUI: The final changes we made to P2PeuR were GUI changes. The changes now display the erasure coding parameters that were used to back up a file in the list of backed up files to the user. This would allow the user to restore a specific file backed up with specific erasure coding parameters. Furthermore we changed the settings window to allow the user to change M, N, block size, and share size, all with sanity checks to ensure proper matches between the parameters.

D. CommApp

After completing our upgrades to support erasure coding, we needed a way to issue commands to P2PeuR from a console so that we could pipe a series of commands to P2PeuR, increasing the simplicity of running tests, and scaling the number of tests that could be run. Without a way to issue commands in a batch, we would need to remote desktop to Emulab and then use the P2PeuR interface manually to login, back up files, retrieve files, and bring peers up and down. This was not trivial since P2PeuR is a native Win32 application and had no mechanism to receive a batch of commands.

Inter-process Message Structure	Supported Commands
typedef struct CONSOLEMESSAGE { int m; int n; int shareSize; int blockSize; char filename[250]; char username[250]; char password[250]; char ip_addr[20]; } consoleMessage;	login <username> <password> add_ip <ip_or_hostname> backup_file <full_path_to_file> restore_file <file_name_only> delete_file <file_name_only> drop_peers <num_peers> drop_data_peers <num_peers> connect_all set_mns <m> <n> <share_size> set_mnb <m> <n> <block_size>

Fig. 4. CommApp/P2PeuR Inter-process communication message structure and supported commands

We decided to take the approach of implementing remote procedure call support into P2PeuR and creating a console application for it, CommApp. This console application could then read a series of commands through its standard input and issue procedure calls to P2PeuR. To implement RPCs we wanted a method that would require the least amount of changes to P2PeuR and provided all of the functionality needed without the complexity of creating, setting up, maintaining, and tearing down socket connections.

After researching several methods we decided to implement a form of inter-process communication using WM_COPYDATA and SendMessage() from the Win32 API. This method will send a message of any length and format to another local Win32 process with handler installed to receive the message.

The first step was installing a handler into P2PeuR to receive messages from CommApp. Doing this was not difficult since the application was already setup to receive and process messages such as message clicks to the system tray icon when the application is minimized to the system tray. To install the handler we used a MESSAGE_HANDLER() call for WM_COPYDATA and created a function which will be passed the data from the messages as they are received.

The second step was actually creating CommApp's functionality and using SendMessage() to pass data to P2PeuR. SendMessage() takes a handle for the window messages are to be passed to, which would be P2PeuR's main window. To get this handle a FindWindow() call is used, which also allows us to make sure P2PeuR is open before trying to send messages to it, alerting the user if it is not. A data structure must then be filled in which is passed to SendMessage(), letting the receiving application know what kind of message it is, such as a login message, or a back up message.

The final step was to pass any data needed by the receiving application to process the message. The data structure we pass and the messages that we have implemented are shown in figure 4. This structure and command functionality allows us to fully control P2PeuR from command line by piping in a series of messages stored in a file for example.

As a brief overview of our commands, *login* will connect the client to the P2PeuR network with the given username and password. The *add_ip* takes an IP address or hostname and will add the host as a peer. The file control commands are *backup_file*, *restore_file*, and *delete_file* which were used during our analysis to back up, restore, and delete files from peers. The *backup_file* and *restore_file* commands displayed the size of the file being backed up or restored, the current system time at start and finish, and the total time it took to complete. The reason we outputted the current system time at start and finish was so that we could match our CPU usage output, which displayed the CPU usage at the current system time, to a specific back up or restore command to analyze the CPU usage during the command. The reason we outputted the total time it took to complete the request and the file size was to compute throughput and network utilization.

The next set of commands, *drop_peers*, *drop_data_peers*, and *connect_all* were used to generate specific back up and restore scenarios for our analysis. For instance, in our analysis we analyze end to end delay which varies with the number of peers connected, as well as the type of data on the peers. The *drop_data_peers* command was used to disconnect from the specified number of peers that held data fragments, to force reconstruction using parity. The *drop_peers* command will drop the newest number of peers passed as a argument to the command and the *connect_all* command will simply reconnect all peers.

The final set of commands dynamically changes the erasure coding parameters, *m*, *n*, *share_size*, and *block_size*. This can be done via the *set_mns* or *set_mnb* commands. The *set_mns* command will send a message to P2PeuR, changing *m*, *n*, and *share_size*, which P2PeuR will then compute *block_size* from automatically to reduce the chance of user computation errors. The *set_mnb* command sets *m*, *n*, and *block_size*, leaving P2PeuR to compute *share_size*.

This functionality implemented into CommApp allowed us to enter a series of commands into a file and pipe them to control P2PeuR, scaling the number of tests we could run on Emulab in a given period of time. Had we needed to use the P2PeuR interface to run tests on Emulab, the number of tests we would have been able to complete would have been significantly smaller. From implementing CommApp we learned about inter-process communication through the Win32 API as well as implementing our own form of remote procedure calls.

V. Experimental Setup

To perform analysis of erasure coding using P2PeuR we chose to use the network testbed Emulab, located at the University of Utah. We chose EmuLab for our evaluation of the system because it offered a set of computers with their own computational and networking resources, which was essential for our analysis. Furthermore, using Emulab allowed us to scale the size of our peer to peer network, offering more resources than we had available to provide for a better analysis. A major concern of the experimental setup was to automate the procedures needed to capture our data as much as possible.

Several scripts were made to completely automate the procedure, allowing us to capture large amounts of data quickly. Scripts started P2PeuR when the nodes booted, generated files of a given size to be transfered during experiments, gave series of commands to P2PeuR through our communication

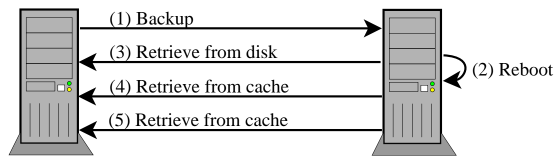

Fig. 5. Data Collection Procedure

application, and moved the data captured to a central location for analysis. Having Cygwin and several GNU tools installed by default on Emulab aided our scripting.

Out of the sets of nodes we acquired on Emulab for a given experiment, one of the nodes was used as our client node and all other nodes were used as peer storage. The next step was to back up and restore files from the network, generating series of commands which were piped to our communication application to automate the procedure.

Considering that our application is file back up and retrieval, one important factor in our results is whether the operating system reads the file from disk or from cache. We could have chosen to only report statistics for one method, to keep consistency in our analysis, but we chose to provide both from disk and from cache results. To do this, the peers that were used as storage nodes were rebooted before file retrieval requests were made. The reason for this is that when the nodes were initially receiving the file data and writing it to disk, some of the data may have been left in the nodes cache. Therefore when a retrieval is requested, some of the data may or may not be in cache. To keep this consistent we reboot the peers to ensure that all reads from the file for retrieval would be initially made from the disk. To get cached reads we performed two subsequent retrievals and recorded the lowest retrieval time value of the two requests. The five step procedure is illustrated in figure 5, where there is one peer being used as a client and one for storage. Our analysis section will further discuss this issue and shows how the cached and disk reads greatly effect the retrieval times.

All of the output generated from our P2PeuR communication application, which was explained in detail under our CommApp section, was automatically redirected in log files and were then used to extract useful data. Another set of scripts were written to analyze the data in bulk, performing functions such as averaging back up and retrieval times, finding the average CPU usage during a series of commands, as well as other functionality found in our analysis section. Overall, putting extra effort into the scripting made the analysis easier and aided in finding important trends in the data.

VI. Analysis

Performing analysis of erasure coding through P2PeuR was done by looking at many different performance aspects of the system, comparing erasure coding to replication as well as cross comparing various erasure coding schemes. This allows us to provide the reader with not only an understanding of when erasure coding or replication out perform one another, but once one method is chosen, what the proper parameters are with respect to different metrics such as end to end delay or overhead.

When a node is referred to as a data peer, we are referring to the fact that all of the fragments on the peer are actually data fragments of the file. A node referred to as a parity peer is referring to a peer who contains only parity fragments of the file. Since round robin placing of fragments is used, when three nodes are in the network, with an erasure coding scheme 2 of 3, the first two peers will only have data fragments, and the last peer will only have parity fragments. This terminology will be used throughout our analysis.

A. Fault Tolerance

In the distributed systems notion, fault tolerance refers to the maximum number of peers unavailable that the system can handle without any serious impact to the integrity of the system. We experimented with various schemes, trying to bring the capabilities of erasure coding techniques to their limits and still quantify the amount of delay this causes. Our figure uses the $m/n/o$ notation, where m is the number of original fragments, n is the number of resulting redundant fragments after erasure coding, and o identifies the type of peers removed from the network. The o parameter with a value of $2d$, for instance, specifies that two data peers were removed from the network between the dispersal of the file and its retrieval. Any value of $1d$ or greater forces reconstruction from parity, since a peer containing data fragments was removed before retrieval. In all scenarios, the block size used was 16KBytes. Our data is represented with bar graphs instead of a line graph since linear increases were observed in every scenario.

Starting with the 2/3 scenario, bringing down a data peer does not affect throughput significantly, since all of the information gets reconstructed using erasure coding techniques in a negligible amount of time. In the 3/5 scenarios, we tested dropping data peers and tested dropping parity peers. Dropping two nodes which carry parity, shown by data point 3/5/2p, does not significantly effect throughput either since the data is still reconstructed from the original data fragments. Dropping two data peers, shown in 3/5/2d, shows that forcing reconstruction from parity has some performance impact, but we believe that it is insignificant. The 9/10/1d scenario also shows similar results.

B. Network Utilization

We define network utilization as the percentage of network resources used during an operation as compared to the link's total capacity. We conducted experiments to find the network utilization for the 100Mb/s links. The throughput per transfer was calculated and divided by the link speed of 100Mb/s to give network utilization. This was done to find out which parameters of erasure coding effect network utilization.

A backup operation was done on all peers, keeping m of n constant at 3 of 5, but changing the share size between $1k$ and $16k$. The total amount of data transferred is the same across all share sizes, however, the amount of data sent in each message varies on the share size which we show effects the

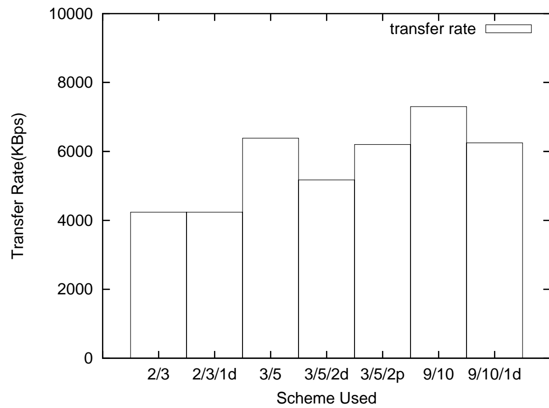

Fig. 6. Comparing schemes with the same level of fault tolerance in respect to throughput with 16KB blocks. The $m/n/o$ notation used refers to number of original fragments, the number of fragments after erasure coding, and #d or #p where # is the number of peers removed and the type is data or peer, d or p respectively.

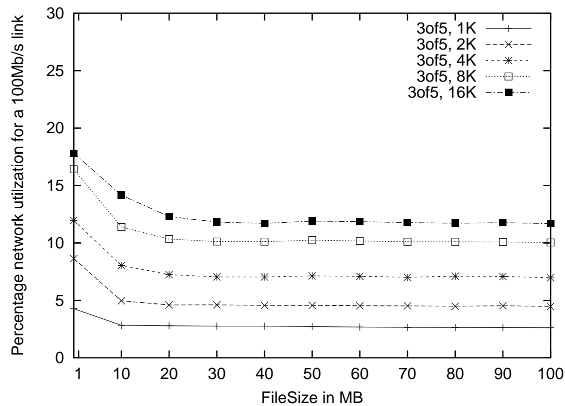

Fig. 7. Shows the percentage network utilization for a single PUT operation, for different share sizes(1K,2K,4K,8K,16K) for erasure coding 3of5

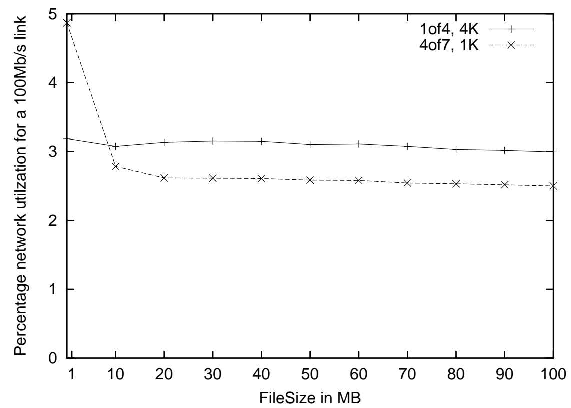

Fig. 8. Shows the perecentage network utilization for replication and erasure coding 1of4 and 4of7, maintaining the same block size of 4K for each, for a single PUT operation

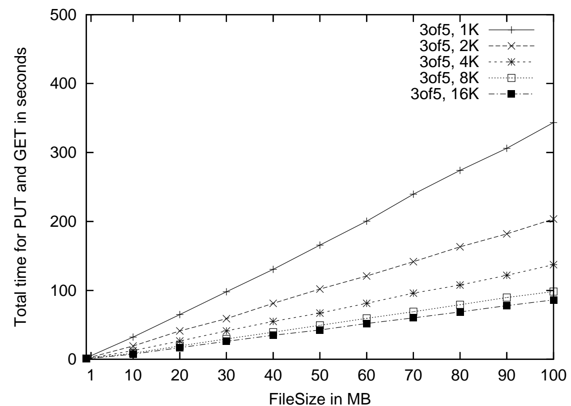

Fig. 9. Shows the end to end time required for storing and retrieving a file with different share sizes(1K,2K,4K,8K,16K) for erasure coding 3of5

network utilization. Higher share sizes will allow more data in each fragment, utilizing the network to a greater degree. This can be seen in Figure 7, where utilization increases and decreases respectively with share size. For lower share sizes, the total number of fragments which must be constructed and transferred over the network increases, increasing the overhead of the system and end to end delay.

In Figure 8, we compare the utilization of replication and erasure coding by varying m and n, but keeping block size fixed at $4K$. The same level of fault tolerance is kept for both schemes, 1 of 4 for replication and 4 of 7 for erasure coding. We show that network utilization does not significantly drop for erasure coding, and is likely an effect of the smaller share size. With network utilization being relatively constant across both methods, we believe that the non-threading and create as you go methodology used by HiSpread which we did not modify, is a limiting factor of the performance of the system. When a file is backed up in both systems, a fragment is created and sent before the next fragment is created and sent. This keeps the network utilization relatively low and could be increased with threading or the construction of all fragments before they are sent.

C. End to End Delay

To measure the end to end delay, we record the time it takes from an initial request for backup using our CommApp IPC tool, and the time P2PeuR responds that it has fully backed up the file. To compare this for a variable factor, we chose share size. From figure 9 we show that that as share size decreases, the end to end delays increase. This follows our results seen for network utilization, as share sizes increases, the network because more utilized. We compared the end to end delay for each filesize represented by the y axis, and specifying the share size used in the labels. The end to end delay for 3 of 5 erasure code with a share size of 1K is is shown to be a magnitude larger than that of greater share sizes. Again, as the share size decreases, the number of fragments that a block gets divided into increases. Thus, we have to send out more fragments, for the same filesize. Sending out more fragments and receiving them increases the overall delay due to the communication overhead. Thus higher shares sizes take less time. However, this holds true only to a certain extent, since the time required for 8K and 16K, seems to be almost similar. So, choosing an extremely large share_size is again not advisable, as the whole purpose of erasure coding will be lost when share size = block size, in other words, M=1 and we thus do replication.

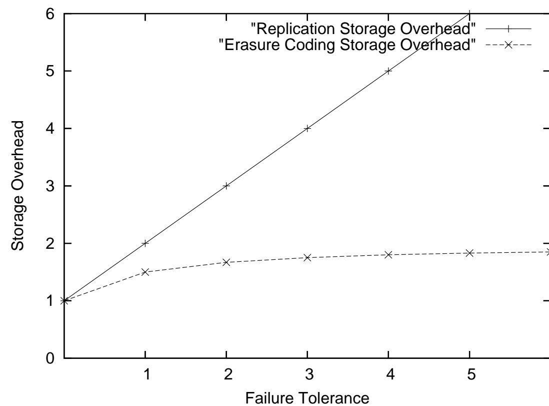

Fig. 10. Storage Factor in replication and erasure coding schemes.

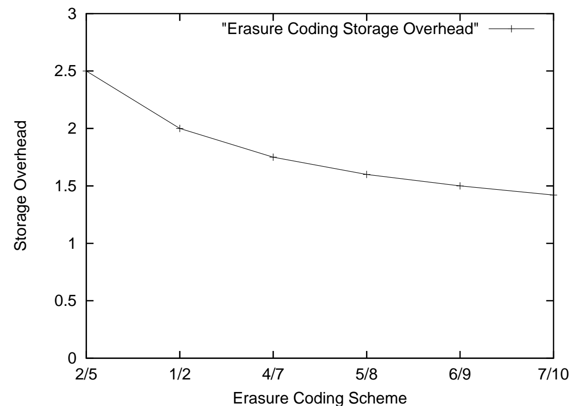

Fig. 11. Storage Factor fluctuation in erasure coding scheme that can handle 3 faults. In the x axis 0.4 represents an erasure coding scheme of 2/5, 0.5 is 3/6, 0.57 is 4/7, 0.63 is 5/8, 0.67 is 6/9 and 0.7 is 7/10.

D. Storage Overhead

In a peer to peer backup system, storage overhead is an important performance factor for the scalability of the system. Storage overhead is defined as the amount of additional storage required to achieve the level of fault tolerance for the given data. For example, with replication of a 1MB file, and a level of three for fault tolerance, the storage overhead is 3MB. In peer to peer backup systems, data is stored on user PCs characterized by limited disk space. Thus, the data redundancy scheme used to ensure that the file is retrievable becomes crucial as it will largely affect the performance of the overall system in terms of storage availability. A peer to peer backup system with high storage overhead constitutes a system with poor disk utilization that results in peers not being able to handle additional backup requests after a short period of time.

Intuitively, erasure coding should perform better in terms of storage overhead than replication with the same level of fault tolerance. One can conclude this by considering the fundamental principal behind each data redundancy scheme. Replication stores the whole file on each peer, imposing a storage overhead that equals the size of the file per additional level of tolerance. In contrast erasure coding does not store the whole on the peer, but only a parity fragment. This imposes a storage overhead equal to the size of the fragment, which is significantly smaller that the size of the file. Our intuition is verified by calculating the storage overhead that various erasure coding and replication schemes impose on the system. Our calculations are depicted in figure 10. For our analysis, we compare replication and erasure coding with equal levels of fault tolerance. We show that as the desired level of fault tolerance increases, replication suffers a significant amount of overhead.

Figure 10 shows storage overhead is increasing linearly in relation to the number of failures you are willing to handle with replication. In contrast, erasure coding has an almost constant storage overhead as the number of tolerated failures increases. For example, a replication scheme of 1/5, represented by a fault tolerance of 4 on the x-axis, introduces a storage overhead of factor 5 for replication, whereas an erasure coding scheme of 5/9 introduces a storage overhead of 1.8.

Along with comparing how the storage factor fluctuates when using various replication schemes against using erasure coding schemes, it is also quite interesting to observe how the storage factor changes when the erasure coding scheme changes, but the number of faults able to handle remain the same. Figure 11 shows the storage overhead factor when the number of faults is kept to 3, and m and n are varied. factor.

From Figure 11, one can observe how the storage factor diminishes as m and n increase. Since n represents the total number of peers the fragments are dispersed across, and m represents the number of these peers that will be tolerated as failures, the fragments are now spread to more peers and thus the storage burden imposed on each peer's storage is smaller.

E. CPU Usage

In this section we would like to show that CPU usage does not increase greatly from using erasure coding. Our analysis was done using a program which logged CPU usage ten times every second and recorded system time with each value so that we could analyze CPU usage during a specific file back up or retrievals. By doing this we could analyze the CPU usage during reconstruction of a file solely from data blocks against CPU usage during reconstructing from both data and parity. If we could show that in the worst case, where all blocks need to be reconstructed using erasure coding, that CPU usage is not significantly greater than we could further build our argument for erasure coding.

Figure 12 shows that, on average, the CPU usage is greater when reconstructing from data and parity, as expected. However, these results also show that reconstructing from data and parity does not have a significant impact processing. The additional amount of CPU usage required for erasure coding fluctuates between 5% and 10% depending on the share size. This supports our argument in favor of erasure coding, since it is shown to not use a significant amount of processing resources which would prevent it from being adopted.

F. Disk vs Cache Performance

Another important aspect of our evaluation was the disk and cache measurements. During every experiment, we performed

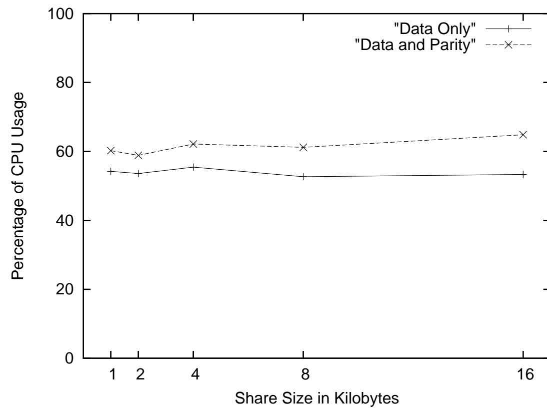

Fig. 12. CPU usage for 2 of 3 erasure coding using various share sizes. The figure shows that CPU usage is greater for reconstructing from data and parity, however we believe it is not significantly great enough to argue against erasure coding

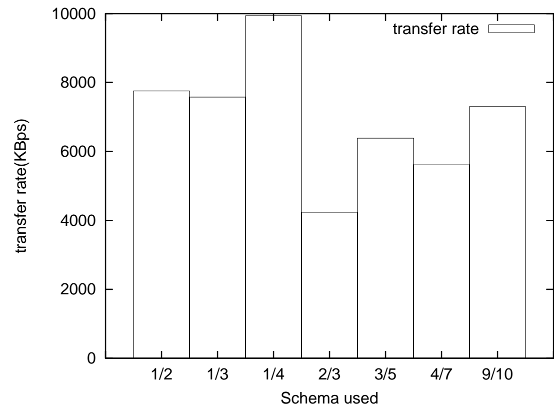

Fig. 14. Scaling out by comparing throughput of erasure coding and replication in 16KB blocks. M/N notation used refers to number of fragments/number of resulting fragments after Erasure Coding is applied.

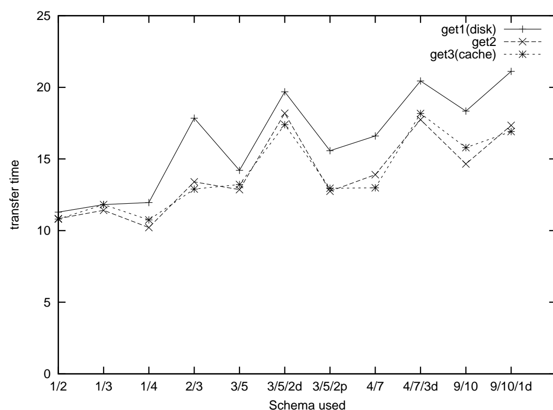

Fig. 13. This figure presents the time required to get a file of 80MB through disk versus getting it from cache. We use the notation described in our earlier analysis results and figures.

the file retrieval method three times. The first of these three was guaranteed to be from disk. To be more confident, we rebooted nodes before this initial retrieval so as to clear any type of Windows cache that may exist from the writes to the disk on backup. The second retrieval done to begin entrance of the data into the cache. The third get was considered as the cache hit which we use for our analysis. In most of the cases, we observed that the third retrieval end to end delay was significantly smaller than the first one, which reinforces our methodology for the experiments. Another important observation we made was that cache retrievals did not seem to affect performance of smaller files.

Our results are shown in Figure 13, where it can be noted with several different peer drop and erasure coding schemes, the difference between disk and cache retrievals is significant. We perform this analysis to support the methodology used in our experiments, of rebooting machines between retrievals and presenting either from disk or from cache results, never presenting a mix of the two in our analysis.

G. Scalability

Testing the scalability of the system, we noticed many interesting remarks regarding Figure 14. In erasure coding the throughput of the system increases, given a level of tolerance, as n increases. This larger value of m would also reduce storage overhead. This scales the system to many peers and provide robustness at a fraction of the disk space needed by replication techniques. Replication techniques on the other hand, can achieve higher throughput. This higher throughput should not be seen through only one point of view. Even though we witness higher throughput in the 1/4 case, we should bear in mind that the overhead comparing to all of the erasure coding schemes used is tremendous. This is the trade-off of the increased throughput. For example in the 1 of 4 case we have 300% overhead space whereas in 4 of 7 we have less than 100%. The scheme 9 of 10 has significantly higher throughput than the rest of the erasure coding schemes and is an indication that the system can scale up to more peers and at the same time increase throughput.

H. Peer Availability and File Retrieval Probability

We have shown that erasure coding can be a better data redundancy scheme in terms of system performance. However, related research has indicated that there is a threshold where replication becomes a better solution than erasure coding. Rodrigues, Liscov [1] and Lin et al[3] claim that the threshold is related with peer availability. Both research papers state that after a series of experiments, erasure coding performs better than replication when the availability of the peers is low. Replication performs better than erasure coding when the availability of the peers is high.

In this section, we attempt to detect the turning point between erasure coding and replication. To do this we arbitrarily determine values that state the probability that a peer is available to service backup requests. Using these values, we calculate the probability of successful file retrieval. These probabilities are calculated for various replication and erasure coding schemes and are plotted against the number of failures that each scheme can handle. It is important to note that

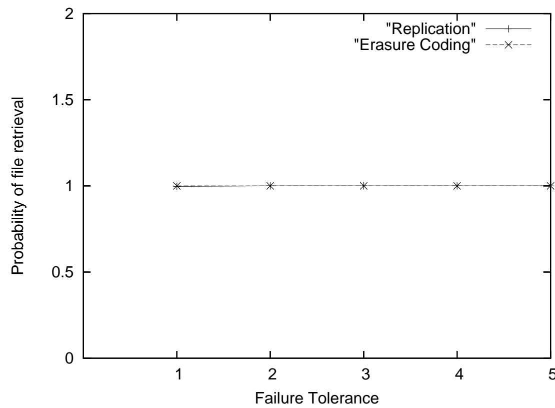

Fig. 15. Successful file retrieval when peer availability is 95%.

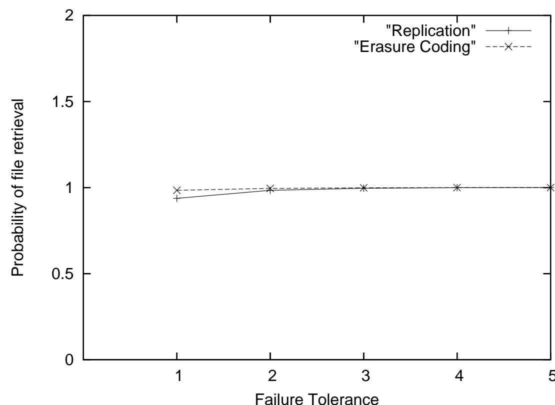

Fig. 16. Successful file retrieval when peer availability is 75%.

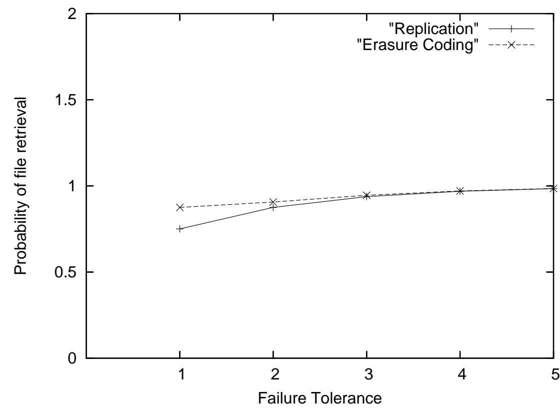

Fig. 17. Successful file retrieval when peer availability is 50%.

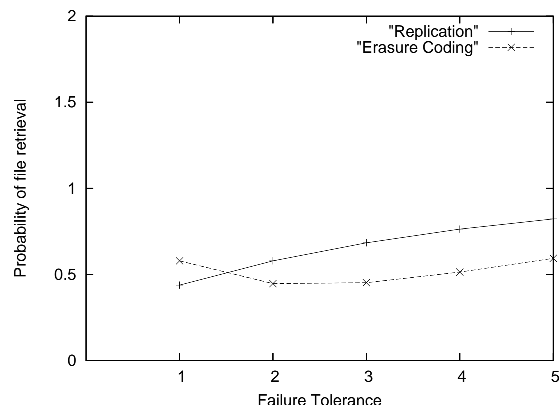

Fig. 18. Successful file retrieval when peer availability is 25%.

each replication scheme is compared against a corresponding erasure coding that handles the same amount of failures, and is the scheme with the highest storage overhead. For example, a replication scheme of 1 of 2 is compared with an erasure coding scheme of 2 of 3 and not 5 of 6.

The probability of retrieving a file for an erasure coding scheme m of n, is computed according to Formula 1 where P_F is the probability of retrieving a file and P_D is the probability for a peer to be down.

$$P_F = 1 - \sum_{i=m+1}^{n} (P_D)^i \tag{1}$$

In the erasure coding scheme m of n, in order to be able to retrieve the file we need any m peers to be available. This is the basis of our analytical approach towards computing the probability of retrieving a file when the data redundancy scheme used is erasure coding. A file can be retrieved if at least m peers are available. Hence the probability of retrieving a file would be equal to one minus the probability of not having enough peers available.

The graphs depicted in Figures 15, 16, 17, and 18 show the probabilities of successful file retrieval when peer availability is set to 95%, 75%, 50% and 25% respectively.

In the first graph, where the availability of the peers is 95%, we observe that replication and erasure coding have the same probability of success in retrieving a file. However in the following graphs as peer availability reduces from 95% to 75%, erasure coding has the advantage over the ability to recover a file. The same event is observed in Figure 17, where peer availability is equal to 50%. This changes when peer availability becomes 25%. In this case erasure coding is preferred only for schemes that handle one fault. For subsequent schemes with a greater number of faults, replication is preferred possibly due to the fact that the larger the number of faults you want to handle, the more peers are needed. The larger the number of peers needed to be available for the retrieval of the file, the more difficult is to have the required number of peers available when peer availability is low. In such cases it is more reliable to rely on a small number of peers than a large number of peers. A set of 10 peers is more likely to be up and running than a set of 40 peers. To handle large number of faults erasure coding requires more peers than replication. A replication scheme of 1 of n requires only one peer to be up in order to retrieve the file whereas, an erasure coding scheme of m of n requires m peers to be up. Therefore in the case where peer availability is so low replication will constitute a more robust redundancy scheme. Hence, when the availability of the peers is high replication can be preferred instead of erasure coding. Determining what constitutes high or low availability is a standard that depends on each system's parameters and should be defined as accurately as possible.

VII. Conclusion

In this project, we analyzed the advantages and disadvantages of using replication and erasure coding. We used several metrics to compare these two methods of distributing data over a finite number of peers. Fault tolerance and end to end delay were two metrics which show the system's ability to scale to many users, and at the same time, face the availability challenges modern peer to peer systems have. An analysis of storage overhead was also done, exploring the correlation between peer availability and file retrieval probability. Interesting remarks were derived by analyzing the resources that are used to implement erasure coding and replication techniques.

Extensive modifications were made to the original source code of Hispread to add the extra functionality required to support the erasure coding library. We tweaked and improved the code further as we found the chances to do so. Writing a command line communication application for P2PeuR gave us the ability to run large numbers of tests. Finally, we deployed a series of scripts to automate testing under different conditions on the Emulab testbed. All of these modifications contribute greatly to the added value of P2PeuR as a peer to peer backup system.

Acknowledgment

We would like to acknowledge Tina Wong, Greg Ganger, as well our project mentor, Michael Abd-El-Malek, for their help and support throughout the project. We would also like to thank Emulab and the Information Networking Institute for providing us with the resources needed to complete our project. We also thank HiSpread for providing us with an open source peer to peer backup system that we could build upon. A final thanks to Carnegie Mellon University's Parallel Data Lab for providing us with the erasure coding library from the PASIS project.

References

[1] R. Rodrigues and B. Liskov, *High availibility in DHTs: Erasure coding vs. replication*. Proceedings of the 1st International Workshop on Peer-to-Peer systems (IPTPS), March 2002.

[2] H. Weatherspoon and J.D. Kubiatowicz, *Erasure Coding vs. Replication: A Quantitative Comparison*. Peer-to-Peer Systems: First International Workshop, IPTPS 2002, LNCS 2429, pp. 328–337, 2002.

[3] W. K. Lin, D. M. Chiu, Y. B. Lee, *Erasure Code Replication Revisited*. Peer-to-Peer Computing 2004: 90-97

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummandi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, *OceanStore: An Architecture for Global-Scale Persistent Storage*. ACM publications 2005

[5] A. Shamir, *How to share a secret*. Communications of the ACM, vol. 22, n.11, pp. 612–613, Nov. 1979.

[6] M. O.'Rabin, *Efficient dispersal of information for security, load balancing and fault tolerance*. Journal of the ACM, 36(2):335–348, 1989.

[7] *http://www.emulab.net*

[8] *http://www.hispread.net*

[9] *http://www.pdl.cmu.edu/Pasis*