Personal Internetworked Notary and Guardian Supported by

Distributed Data Storage

by

Stephanie Hope

Submitted to the Department of Electrical Engineering and Computer Science

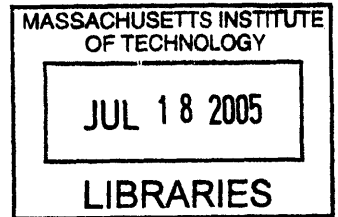in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 19, 2005 [June 2005]

Author_____
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by_____
[Supervisor's Name]
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ARCHIVES

Personal Internetworked Notary and Guardian Supported by Distributed Data Storage

by

Stephanie Hope

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 2005

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Electronic medical records (EMR) have the advantage of being easy to modify, organize, and distribute and would give healthcare providers a more complete patient history. The ideal EMR system should maintain records that are easily transferable and are administered and owned by the patient. One such system striving to meet those needs is Personal Internetworked Notary and Guardian (PING). A system like PING would run off a single server. The problem with centralized storage is that it creates a single point of failure which has the potential for problems. I propose a distributed storage system that is similar to RAID 5 system. This Distributed RAID System provides reads at approximately 1MB/sec. Writes however are slow, but could be improved by running the Distributed RAID system on different hardware.

Thesis Supervisor: Peter Szolovits
Title: Professor of Computer Science

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

First, I would like to say thanks to Professor Szolovits for being such a great thesis advisor. His guidance made the best of what was a difficult journey through this thesis process. When my original thesis plan fell through, he was patient and understanding while I did a major redesign. I would also like to thank Deshen Moodley for helping to guide me toward my thesis topic.

Thank you Fern for getting me into yoga to relax me and dodgeball to rev me back up. Neha, you were the best desk mate ever and you definitely made working at my desk more fun. Special thanks to the gnomes that live in my computers--because what didn't kill me has made me stronger.

I would also like to thank my friends and family, because having a support network is vital. Thanks to Jennie and Miquela for helping me through marathon work sessions when I would have preferred to hide in my bed. Thank you Bo for your unwavering confidence in me. Mom, thank you for helping me so much during this whole process. I know at times I have driven you crazy, driven up the wall and driven you 'round the bend--but I could not have done it without your constant support and encouragement. I am so blessed to have you as a mother. Dad, thanks for cracking the whip. And Dad, yes it's done now.

Introduction

Currently, medical records are stored on paper and split between the various institutions that created them. For example, when you visit your primary care physician, he keeps a record of your visits with him. These records however, are often separate from those that you may accumulate through various hospital visits. Current medical knowledge and treatments have allowed modern medicine to extend life. These treatments, however, are the most successful when all patient information is known. When a patient's medical information is spread between various doctors and hospitals, collecting this information can be very difficult.

Ideally, these dispersed records would be united. Uniting them on paper however, would be very difficult. The best way to accomplish this would be to move medical records to an electronic form. Electronic medical records (EMR) have the advantage of being very easy to modify, organize, and distribute. The use of EMRs would give healthcare providers a more complete patient history.

If implemented in a manner that gives ownership of the record to the patient, EMRs can have the potential for additional benefits. Currently doctor, hospitals and HMO's have control of a patient's records. Each keeps its own portion of the medical record which is often not accessible to the patient. Patient controlled EMRs would finally allow patients to own their own data. With ownership, comes the possibility to make patients more involved in their own medical treatments.

Many individual institutions have deployed, or are in the process of deploying, their own EMR system. Sutter Health plans to deploy an electronic health record system that will be utilized by 27 hospitals, 6 medical foundations and 11 independent physician associations in Northern California[1]. Set to start in 2006, this system will not offer patients direct control of their EMRs, but will allow patients secure access (SSL) to their record via the internet and the ability to interact with their medical care givers online.

A similar system, PatientSite, is currently in place at Beth Israel Deaconess Medical Center. PatientSite provides patients secure internet access to portions of their medical records. The site also allows patients to ask questions about care and symptoms, set up appointments or referrals, and renew prescriptions[2]. Although it too does not offer

direct patient control of the record, it does allow patients to add comments to portions of their record.

PatientSite and the Sutter Health system are just two examples of a growing trend in providing patients with access to their medical information. Although this is a benefit to the patient, these blossoming systems present a new problem, portability. Patient mobility, changes in location and changes in employment/health plan, result in a change in health care provider[3]. If the EMRs of different institutions are not compatible, then patients will face the same fragmented record problems that paper medical records posed. Electronic medical record portability is necessary in order to make patient information available to all healthcare professionals, regardless of affiliation.

In summary, the model EMR system should maintain records that are easily transferred between institutions and that are administered and owned by the patient. One program that strives to fulfill these goals is Personal Internetworked Notary and Guardian (PING). PING was developed out of a joint effort between Children's Hospital Informatics Program and CSAIL's Clinical Decision Making Group. This program allows individuals control over their medical records by allowing them to decide who is authorized to view and update their record. PING allows patients to define a variety of access rules. Rules can have a broad scope (let all doctors modify my record) or a narrow scope (only allow the psychiatrist at Glenn Oaks to view the portions of my record dealing with mental health). It also gives patients the ability to add comments to entries made by healthcare providers and add their own entries too.

Ordinarily a system like PING would run off a single server that implements a redundancy mechanism (usually RAID). However, if the goal is to allow all of the people in an entire country to manage their medical records, then a new data storage method must be found. The problem with centralized storage is that it creates a single point of failure. If there is a hardware problem with PING, then the entire system (and by extension all of the medical records) become inaccessible. Similarly, if some catastrophe destroys the building that housed PING, all current data would be lost. The system would then have to be restarted from a new location, with data from the last data backup (assuming the data is periodically backed-up). Depending of the extent of the problem, the time needed to restart the system could be excessively long.

Implementing PING with distributed storage would address many of these problems. Initially I hoped to provide PING with distributed storage by using OceanStore. OceanStore is a distributed hash table that allows nodes to join and leave the system freely while still guaranteeing persistent, distributed data storage with serializable management. OceanStore however, is a project that is currently underdevelopment and, as such, it was not feasible to work with the non-stable releases. Therefore I proposed a distributed storage system that is similar to RAID 5. However, instead of storing all of the disks in one physical location, I suggest a design that places each disk in a separate location. This suggested system would have all of the reliability and benefits of a RAID 5 system, without the dangers that accompany storing the data in one central location. By separating the disks, it becomes much harder to obliterate all of the data. In addition, I suggest a design that does not depend a on a centralized controller so that the core PING code can run from any server.

The rest of this paper is organized as follows; first, a description of the technologies that were used, including PING and RAID, followed by a discussion of related work. Next is a detailed account of the design of the Distributed RAID system along with an explanation of the design decisions. This is followed by an overview of the test conducted on the system and the results yielded. Finally, I conclude with a discussion of the test results and an overview of future work to be done.

Background

PING

The goal of PING is to empower patients by providing them with the tools to control the dissemination of their medical information. It aims to help them unite information that hospitals, and other organizations, might not want to share for proprietary reasons[3]. Patients designate access policies to control who has the right to view and modify their records. The patient record itself is a group of HL7 compliant XML objects. Each document in a PING record is wrapped in XML that provides meta-data about that object. This meta-data includes its classification in the PING record, a description of the contents, its creation time and its modification time[5]. The meta-data schema of the PING records is designed to be easily incorporated into existing and future standards, but not reliant on them. Since the PING schema is made available to the public, organizations that chose not to use PING can still load PING records into their systems and export the appropriate PING document when they are done providing medical services.

Client

Figure 1 shows the components of PING. Interaction with the system begins with the PING client, or client application. Current PING client applications include Java applications that provide a browseable view of the record and record policies maintenance, and a Fax server that inserts documents into a PING record. When communicating with the PING server, client applications send it XML messages to the PING server. The only restrictions on PING clients are the use of the PINGTalk protocol, authentication protocol and authorization protocol.
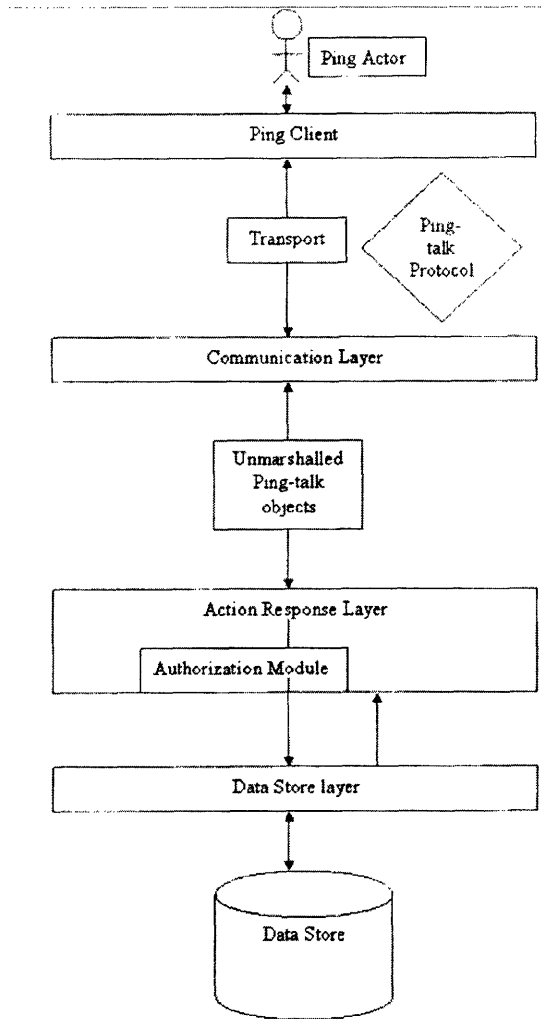
Figure 1: PING architecture overview[5].

A typical PINGTalk request contains a session ticket (optional), the actor making the request, the action to be performed, and credentials needed, if any, to perform the action. The request is formed into an XML object and sent to the PING server. The PING server responds with an XML object that includes the result of the action along with outcome codes indicating the success of the action. The schema of both the request and the response are derived from the PINGTalk schema.

The Authorization Module uses actor session (authorization) information along with requested actions to determine whether or not an actor has the permission to perform the requested action on the record or record element. Actors are authorized to perform actions based on their role, group membership and/or proxy representation. Roles have associated privileges such as the ability to create records, read records and add or update elements in records[5]. Agents may have multiple roles, but they must choose one when logging into the PING system. This role is the one that will be used by the Authorization module. Although an actor may have the privilege of updating elements in a large set of records, individuals whose records fall into that set can still revoke this privilege through patient-defined access policies. Actors are also granted permissions through group memberships. Group memberships do not have any associated privileges--but patients can grant or deny actions based on group memberships. When an actor acts as a proxy for another actor, it gains the access policies for that actor.

This unique form of access control eliminates fixed hierarchies of access levels, an important quality in a model which needs to express a patient's variable access preferences based on the type of individuals that might want to view the patient's data. It is vital to patient privacy that patients have control over who reads and modifies their records[4]. The PING server access control is similar to Role-Based Access Control (RBAC) except for two main differences[5]. First, PING does not rely on a fixed hierarchy like RBAC. Second, while both systems allow administrators to assign roles and corresponding privileges, PING allows patients to override these privileges with their own set of policies based on specific actors or groups of actors. PING first checks the actor's role to see if the actor has the privilege to perform the action. If it does not, then the actor is not authorized. Next, it checks the policies specific to the actor's identity, policies specific to his proxy and policies specific to his group membership--in that order. The result of the authorization is the result of the first set of policies encountered[5].

The three main layers of PING are the Communications Layer, the Action Response Layer, and the Data Storage Layer. The Communications Layer provides secure HTTPS communication for the marshalling and unmarshalling of XML objects. After unmarshalling a XML request from the Ping client application, the Communications Layer sends it to the Action Response Layer. When it receives a response from the Action Response Layer, it marshals the object back into XML and returns it to the client application. This layer is also responsible for processing user configurations and initializing other modules. The current implementation of PING uses a Java Servlet--however, like the Ping client application, this too can be replaced by another application framework as long as it complies with the PINGTalk protocol[5].

The Action Response Layer performs the logic necessary to process particular actions. It also maintains actor session information which is used by the Authorization Module to make access decisions. To respond to an action request, the Action Response Layer forwards the actor's session information, along with the requested action, to the Authorization Module. The Authorization Module determines if this action is allowed by the actor and returns the result to the Action Layer. If the actor is authorized to perform the requested action, the action response layer generates the appropriate result. The action response layer also utilizes a configurable data store for obtaining the record information needed for the result[5].

The Data Store Layer is responsible for storing patient records. The PING Server implementation allows for a wide variety of back-end storages. The only requirement is that the data storage be able to implement the proper interfaces. The current implementation of the PING Server uses a single server as the data storage.

Although this method of data storage will support thousands of patients, it does not currently scale to millions of users. If the goal is to provide EMRs for an entire country, a different implementation must be considered. A possible fix to this problem would be to keep the idea of central storage but to spread the information over several disk using a RAID system. Although this increases the amount of raw data storage, it has the same single point of failure weakness that the current implementation does.

RAID

RAID stands for Redundant Array of Independent Disks. RAID systems are composed of a single computer and multiple magnetic disk drives connected to that computer. A magnetic disk consists of a stack of platters which stores information in arcs called sectors. The disk actuator controls the movement of the arms while the arm heads read from and write to sectors of the platters[6]. A complete ring of sectors is called a track and a vertical stack of tracks at the same distance from the center is called a cylinder. Figure 2 shows these components.
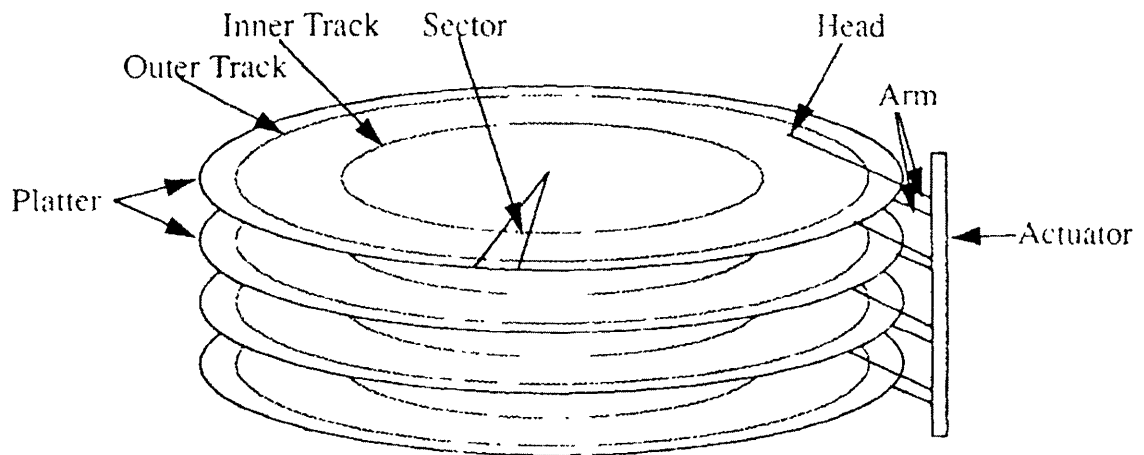
Figure 2: Components of a magnetic disk drive[6].

The purpose of a RAID system is to provide a mechanism of redundant storage. RAID systems come in different levels, each level provides information redundancy in different manners. Redundancy is provided through mirror, data encoding or a variety of both. With mirroring, data is simply copied to an additional disk. When a disk fails, the information is retrieved from a backup disk. The original disk can either be repaired with the information in the backup disk or the entire disk can be replaced. When RAID systems use encoding, the file is broken up into pieces, some of which can be used to recover from damage to the file. There are two ways to encode a file, Hamming encoding and parity encoding.

Hamming encoding algorithms divide the information into code words--a series of 1's and 0's. These code words are then either restricted in value or supplemented with additional bits in order to detect and/or correct for errors in storage. The Hamming distance is the smallest number of bits that have to be changed to turn one code word into another. Take for example an encoding scheme where the code words are **101100** and **100110**. In this case, in order to mistake one of these code words for the other, at least 2 changes would have to be made--therefore the Hamming distance is 2. When the Hamming distance of a code is d, "it is possible to detect d-1 error and correct $\lfloor (d$ - $1)/2 \rfloor$ errors"[7].

Another way to perform Hamming encoding is to intersperse the code word with error detecting parity bits. Specifically, the parity bits are placed at the power of 2 indexes. The parity bit at any location encodes for the information of all blocks whose decomposed power of 2 sum includes that number. For example, 7 decomposes to 4+2+1. Therefore the parity bit at index 1, 2 and 4 will each encode information for the bit at index 7. To encode this information, the XOR operator is used. Each parity bit is the XOR of the bits whose decomposition sum includes them. For example, in a code word of 7 bits, bit 1 is the XOR of bit 3, 5 and 7.

The second way to encode a file is to use parity encoding. This extends the idea of Hamming encoding with parity bits by placing the parity bits in a separate block. In this scheme, a file is broken down into **m** blocks. A parity block is then calculated by doing a bitwise XOR of the **m** blocks. Now if any 1 of these **m + 1** blocks are lost, it can be recovered by XOR the remaining **m** blocks.

Each RAID level (with the exception of RAID 0) uses either mirror, encoding or a combination of both to provide data redundancy. The most common six RAID levels are discussed bellow.

RAID 0

RAID 0 is the only RAID level that does not supply redundancy. In a RAID 0 system, data is broken down into blocks and each block is written to a different disk--this is also known as striping. Since RAID 0 offers no redundancy, a single block error eliminates the entire file. Similarly, when any single disk goes down, the entire storage becomes unreadable.

RAID 1

RAID 1 provides redundancy through mirroring. In this storage scheme, half of the disks in the system are used as mirrors, or copies of the information. This means that the overall system has 1/d as much storage as a non-redundant system, where d is the number of disks in the system. When information is written to a RAID 1 system, the data is copied to every disk. For reads however, the information is just read from a single disk. The benefit of this system is that it is able to service many reads because every disk has the information. For this reason, RAID 1 systems are often used for databases6.

RAID 2

RAID 2 uses Hamming codes to provide redundancy. In RAID 2, the number of redundant disks is proportional to the log of the total number of disks in the system6. Data is bit striped across the non-redundant disks and Hamming encoded bits are written to the redundant disk. The benefit of this system is that the Hamming codes used are resistant to multiple faults/errors. This is better than many other RAID levels which can only recover from a single error. The downside of RAID 2 is that the calculation of the Hamming code is CPU intensive.

RAID 3

RAID 3 also uses bitwise striping. However to provide redundancy it only uses one additional disk. This contains the XOR (parity encode) of all of the other bits in that stripe. This configuration can only recover from a single failure. Write requests access all of the data disks as well as the parity disk. This can create a bottleneck at the parity

disk6. Read requests access all of the data disks--the parity disk is only accessed if the data is missing.

RAID 4

RAID 4 is very similar to RAID 3--except the data is stripped blockwise instead of bitwise. The length of the block is configurable. Just as with RAID 3, data is distributed across the data disks and there is a single dedicated parity disk. The blocks on the parity disk are computed by computing the bitwise XOR of the blocks on the data disks.

RAID 5

RAID 5 also does blockwise striping; however there is no dedicated parity disk. Parity blocks are interspersed with data blocks. There are two ways to achieve this, left-symmetric or right-symmetric. Figure 3 shows the left symmetric block layout of RAID 5. RAID 5 "has the best small read, large read and large write performance of any redundant disk array" 6. It does well with small and large reads because all disks in the system are used to service the request, unlike systems with a dedicated parity drive.

1	2	3	Parity
4	5	Parity	6
7	Parity	8	9
Parity	10	11	12

Figure 3: RAID 5 with left symmetric block layout.

RAID 6

RAID 6 is very similar to RAID 5. The critical difference is the method of error encoding. RAID 6 uses an error correcting code (Reed-Solomon codes) that is capable of recovering from two disk failures. The error correcting codes used in RAID 6 are often expensive to calculate (with out specialized hardware). The expensive parity calculations, combined with the drawback that RAID 6 parity information requires more storage than other RAID levels, prevent many people from implementing RAID 6.

Related Work

Although RAID systems are usually implemented with hardware, there are RAID systems that have been built through software. This work has led to RAID-like systems where instead of having all disks in the array in one location they are physically separated. One such system, Federated Array of Bricks (FAB), is in development at HP Labs.

FAB

FAB seeks to supply scalable, reliable, low-cost, distributed storage. The FAB systems are composed of multiple bricks--"small rack mounted computers built from commodity disks, CPU, and NVRAM"[8]. Clients send I/O requests for a logical block[i] to a brick which acts as the connection point to the rest of the system. This eliminates the need for a central controller in FAB.

Each brick in the system maintains the following data structures: a volume layout, a seggroup, a disk map and a timestamp table. The volume layout and the seggroup act as the bricks global metadata[8]. They allow the brick determine which bricks store which logical blocks. The disk map contains the mapping between the logical blocks that a disk stores and their mappings to physical memory. The information in the timestamp table is used to validate information stored on the brick.

Since all bricks run the software and maintain the same data structures, clients are not bound to a particular brick[8]. The chosen brick, or coordinator, then contacts other bricks (storage bricks) in the system requesting that they perform the desired operation on the logical block. The storage bricks then map the logical block to physical memory and modify or return the corresponding data. Because of this scheme, FAB does not need a central controller. Figure 4 shows the sequence of events that transpire when locating a logical block. In the upper half of Figure 4, the coordinator brick attempts to locate 1KB of the logical block *<volume-id, 768KB>*. The volume layout and volume table specify seggroup **B**, **D**, and **E**. The lower half of the diagram shows how the request for the information is handled by the storage bricks. The Disk map transforms the logical block to a physical disk and offset on the storage brick.

---

[i] Logical blocks consist of a volume-id and offset.
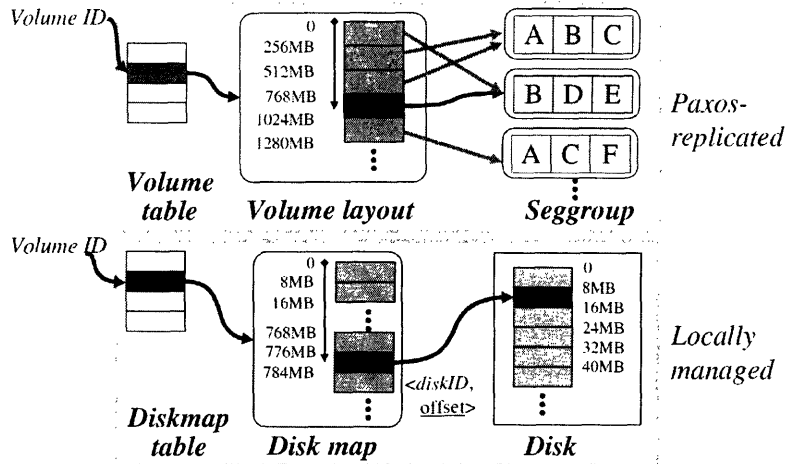
Figure 4: Data structures inside of a brick. Also shows how the data structures are accessed in order to locate a block[8]. The upper half of the diagram shows the flow of information in the coordinator. The lower half of the diagram shows the flow of information in each of the bricks that store requested information.

FAB implements two voting based methods of redundancy, replication and erasure-coding. To implement replication during writing, data is time-stamped and multiple copies are distributed to bricks. When a read occurs, the coordinator returns the data that a majority (quorum) of bricks believe to be the newest data, based upon timestamps. If the coordinator receives response that does not agree with this quorum, it re-writes the quorum agreed upon data back to the entire system, with a new timestamp. To increase the reliability and redundancy of the overall system, FAB also uses erasure-coding. Similar to the replication scheme, parity blocks are generated and time-stamped on writes and replaced with new information if stale on reads.

Federated Array of Bricks has many similarities to the Distributed RAID system developed for this thesis. Both systems lack a central controller and use a method of encoding to provide redundancy. Distributed RAID however, takes a more aggressive approach to recovery by periodically checking files. FAB however, recovers files only when reads reveal a fault. The Distributed RAID system is described fully below.

Methodology

Distributed RAID System

Overview

I have written a system that expands on the software RAID idea. In traditional RAID systems the disk arrays are in the same physical location. Normally, a system like PING would use a RAID system to provide redundancy for the information that it stores. Having all of the files stored in one place however creates a single point of failure--if the system hosting the PING Server goes down, all files become unreadable. In order to eliminate a single point of failure however, storage can be distributed.

The focus of this thesis was to build a distributed storage prototype for PING. This storage prototype implements many of the same design principles as a RAID 5 system, but places the disks in separate locations. In RAID 5, the computer specifies a logical block number and the controller, located on the disk, translates it into a physical cylinder, track and sector. The computer (central controller), which issues I/O requests, implements a locking system in order to prevent simultaneous reads and writes. When an error is detected in a RAID system, the central controller coordinates repair.

The Distributed RAID system has all of the functionality of a RAID 5 system, but achieves it through different means. Similar to RAID 5, the individual storage devices of the Distributed RAID system keep track of where files are stored. However, Distributed RAID does not achieve locking through a central controller. Instead, it uses the storage devices of the system to coordinate the locking of a particular file. Although coordinating file locking between multiple storage devices requires more processing than checking a single data structure (in a RAID system), it has the advantage of not requiring a central controller. Because it lacks a central controller, repairing data must also be coordinated between the storage devices.

16

Like RAID 5, Distributed RAID is designed to tolerate the loss of a single disk. In terms of a specific file, both RAID and Distributed RAID can recover from one fault per stripe.

While a RAID 5 system is composed of a single controller and multiple magnetic disks, Distributed RAID consists of a Client and multiple storage devices. The main components of a storage device are: the Disk, physical storage, the LockManager, the FileDatabase, the FileChecker, the ConnectionChecker and the RestarterService. The Client, which sends I/O requests to the storage devices of the system, is most like the computer in the RAID 5 system. The difference, however, is that the Client stores no file information. There is nothing in the design of the Distributed RAID system that prevents the use of multiple Clients. The Disk is the only remote object that the Client has access to and is most similar to the magnetic disk of a RAID 5 system. The Disk is responsible for retrieving data and storing data to physical storage in response to Client requests. It also forwards requests to the LockManager, when appropriate.

The LockManager and the FileDatabase are both responsible for keeping information on the files in the system. In RAID 5 systems, this information is usually kept in the central controller (computer). The LockManager keeps track of which files are currently being written to and read from; while the FileDatabase keeps track of the total size of the file, not just the size of the portion being stored in physical storage. The FileChecker, that checks the integrity of the files stored in physical storage, also performs an operation that is handled by the central controller in a RAID system.

The ConnectionChecker and the RestarterService help the Distributed RAID system detect and recover from a Disk that has gone down. The ConnectionChecker periodically ensures that the other Disks are up and if not, restarts them. The RestarterService is the component used to restart the Disk if it has gone down. Inter-Disk connectivity is not a concept that has a parallel in RAID systems. The magnetic disks of those systems do not communicate with one another.

Disk

Function

Each Disk knows its host name, the port where its class files are available and its offset. The offset tells the disk which Disk it is in the overall system. Disks also contain physical storage. In the implementation of this thesis, the physical storage is a dedicated directory on the computer. The Disk has two different modes: read-only and normal. In read-only mode the Disk will not accept any request to write to files. In the normal mode, the Disk is capable of receiving all categories of requests. When the Disk receives lock requests, they are forwarded on to the LockManager. Read/WriteRequests are handled by the Disk itself. ReadRequests consist of the name of the file to read and the key for this ReadRequest (see LockManager). WriteRequests consist of the name of the file to read, a key, and the size of the file (including parity blocks) to be written.

When a Disk receives a ReadRequest, it first checks with the FileManager to verify the key. It then accesses the FileDatabase to determine the size of the document to be read. It uses this information, along with its Disk offset to determine which blocks it *should* be storing on its physical storage. The Disk reads these blocks in from physical storage and then packages them up into a response. When a Disk receives a WriteRequest, it also checks with the LockManager, but it uses the FileDatabase to update the known size of the file before writing the blocks out to physical storage.

Security

A Disk implements several security features. First, it uses a java policy file to restrict the directory to which the Disk` is allowed to write. The Java RMISecurityManager (required to instantiate the Disk) enforces policies in the java.policy file. It also increases security through design by restricting the available methods that users can call. The only method visible to the Client is the *executeTask*

method, which takes as an argument a request and returns a response. In this way, the Client has no direct access to the physical storage or other methods of the Disk.

Limitations

The Disk performance is strongly tied to system performance and resources. The number of concurrent calls that can be made to the Disk are determined by the number of available sockets, the number of available threads and the amount of free memory[9]. The amount of memory roughly needed to run the Disk (assuming File Checker is running) is:

$$= \text{maximum file size} + \text{number of concurrent users} \times \frac{\text{maximum file size}}{\text{number of Disks}}$$

The machines used in running experiments for this thesis have a maximum amount of memory that they can devote to the JVM, 2000MB[10]. This restriction comes from a limitation of the JVM for Linux (32-bit processor). On other platforms, the amount can be twice as much, or even utilize all of the system resources. Using a maximum file size of 70MB, memory allocation of 2000MB[i] and number of Disks of 3, the number of concurrent users (maximum number of threads) comes out to 74.

FileChecker

Function

The FileChecker is a scheduled task that runs when the Disk activity is low. Run in its own thread, the FileChecker acquires a read lock and attempts to read in a file chosen for it by the FileDatabase. It attempts to gather blocks from the other Disks along with how long those Disks believe the file to be. The first step in the file checking process is to determine how long the file should be. When returning the blocks for the file, each Disk also includes the size of the file retrieved from the FileDatabase. It then uses these potentially different sizes to determine the actual size. A size that has **d -1**

---

[i] Of the 2000MB, some must be allocated to overhead. This amount was estimated to be 50MB

votes (where **d** is the number of Disks), or more, is considered a correct size. If no such size exists, then the file is deemed uncheckable. The system can irreconcilable sizes when a Client crashes in the middle of a write before it has had time to write to **d -1** Disks. If it can correctly determine the size of the file, the FileChecker then begins checking the blocks for that file. For each block on the Disk, the FileChecker examines the other blocks on that row (stripe). If there are any blocks missing on the remainder of the stripe, then the local block is assumed to be valid. This is because with so few blocks, it would be impossible to verify the block content. If no blocks are missing, the FileChecker calculates the XOR of the other blocks and compares it to the local block. If the two are not equal the file is considered corrupt and the check fails. If the local block is missing, then it is replaced with the encoded block. Once all local blocks have been checked, the FileChecker then obtains a write lock (only for the local Disk) to write out new versions of missing local blocks.

Limitation

One of the main limitations on the FileChecker stems from the Distributed RAID System's emulation of RAID 5. RAID 5 systems can only recover from a single error and have a granularity of a single block. In other words, a RAID 5 system can detect and recover from a single missing block, but it cannot recover from individual byte errors on the block (although it can detect it). Take for example the situation where all blocks on a stripe exists, but the local block does not equal the XOR of the other blocks on the stripe. A RAID 5 system (along with Distributed RAID) can detect this error, but cannot identify if the local block or one of the remote blocks has a byte error.

Another limitation comes from the resource utilization of the FileChecker in the computation it does to calculate the parity block. Reading in an entire file uses up memory of the JVM. Every file being checked is equivalent, in memory, to servicing **d** ReadRequest: where **d** is the number of Disks in the system. For this reason only one file is checked at a time.

LockManager

Function

The LockManager is an in-memory data structure that keeps track of what files are being read from and written to. It stores two types of locks: read locks and write locks. A read lock consists of outstanding tickets and associated timestamp, while a write lock consists of a single ticket and timestamp. The LockManager also keeps its own connections to other Disks in the system to help coordinate the locking process. All requests to read or write to a file must use a key which is supplied by the LockManager. This key is not meant to secure the data transaction, but instead to ensure that clients can only access the information they requested. Before reading or writing to a file, the client will send a lock acquisition request to the Disk, which forwards the request on to the LockManager. The LockManager then creates a ticket for this request by combining the client identifier, the file name, and the time of the request. The LockManager then checks if the file is available for locking by examining its own table.

When attempting to acquire a write lock on a file, the LockManager first checks locally to see if there is an existing write lock. If there is, it examines its timestamp to see if it has expired. If the write lock has not expired then the operation fails and returns null. If there is not a write lock, but there is a read lock for the file, the Lock Manager traverses the ticket timestamp pair, removing expired ones. If after this process, there are no more mappings in the read lock, the LockManager acquires a local write lock on the file. Similarly, when determining if it can acquire a local read lock the LockManager checks to see if there is an existing write lock that has not expired. In either case, if it is able to lock it locally, the LockManager then contacts each Disk in order to acquire the appropriate lock on the file: if it is available. The order that it uses is that of increasing Disk offset. So that if there are 3 Disks and the LockManager on the second Disk is attempting to acquire a lock, it will first contact Disk 3, then Disk1. If a Disk refuses a request to lock a file, the LockManager cancels the locks in the order that it acquired them. If the LockManager is able to acquire a lock on all of the Disks it returns the ticket (via its Disk) to the client.

21

The current implementation of the system is susceptible to "flood" attacks. A malicious user could harm the system by flooding it with read requests for files. If there are many read requests on a particular file (malicious use), the locking protocol described above will tend to slow the system down when it starts to receive write requests for that file (normal use). To obtain a read lock, one need only establish that there is no existing write lock; or that if there is, it has expired. Obtaining a write lock however requires that every outstanding read lock on a file be examined to determine if it has expired. This leads to a situation where read locks are continually acquired on a file, while write locks spend an increasing amount of time checking the validity of each read lock. What stops this from happening is that the LockManager implementation acquires a synchronization (resource) lock on the internal lock table whenever it examines it. This prevents more read locks from being acquired on the file, and gives other ReadRequests time to complete. However, this also prevents locks on different files from being created, slowing the system down as a whole. In other words heavy demand for any particular file can drag the system down.

One way to solve this problem is to copy the data structure for any particular file found when examining the lock table. These locks can then be examined to determine if any of them have expired. If they have, a resource lock can be re-acquired to remove the expired locks and possibly add the desired one. This option has a high overhead cost and will increase the work of the garbage collector.

Another option would be to remove the timestamp component from the read and write locks. This would greatly decrease the amount of time needed to obtain a lock because the existence of any lock would determine the outcome of the acquisition process. The drawback to this option is that if a client crashes, and leaves an unused lock in the LockManager, there would no longer be a way to distinguish between new and very old locks. Abandoned locks would then prevent anyone else from reading the file (in the case of write locks) or writing to the file (in the case of read locks).

Although the locking algorithm presented above will not deadlock, it is vulnerable to livelocking. Livelocking occurs when two actions compete for the same resource, fail, abort their action and then retry in unison. The danger of livelocking is that each process appears to be doing work, but neither will every complete[7]. One prevent livelocking is to use exponential random backoff. Exponential random backoff increases the likelihood of successful lock acquisition by increase the amount of time before a retry by a random amount.

Another alternative design for the LockManager would be to implement two-phase commit. Although two-phase commit does not necessarily address the problem of livelocking, it does address the problem of an incomplete write due to Client crash mentioned above. In two-phase commit, each node keeps a log of values and actions that have been taken. A coordinator node controls changes to a file by first checking with the other nodes (subordinate nodes) to determine if they are willing to perform the commit. If it receives the appropriate number of affirmative responses it then sends each node a message to commit the data to disk, otherwise it sends each node a message to abort. When the nodes receive the commit message, they commit the data and then notify the coordinator of their success. During each phase of two-phase commit, the node logs its actions. If the coordinator fails before the checking whether to abort, its log helps it restart the process. If the coordinator fails after sending the abort/commit message, the other nodes have all the knowledge they need to complete. If a subordinate fails before the commit/abort message the coordinator can send an abort message to the other subordinates (after waiting a specified time). Subordinates that fail before receiving a commit/abort can use their logs to contact the coordinator for the status of the operation.

FileDatabase

Function

The FileDatabase keeps track of all of the files that have blocks stored on the Disk, and the total number of blocks for the entire file (including parity blocks). The values in this database are updated when blocks are written to the Disk and used to determine which blocks should be on the Disk when file data is requested. The FileDatabase is backed by an implementation of Berkeley SleepyCat DB JE Edition. SleepyCat databases comply with the ACID semantics and transactions. It also supports full recoverability of written database entries. Since all changes to this database are done through committed transactions, once an entry has been written to the database, the FileDatabase will always know about it. Although committing after every transaction creates its own performance penalties, it does limit the number of transactions that need to be recovered, should the system crash between writing to the log and flushing to disk. In addition, as the size of the database grows, it will be efficiently accessible. This is another improvement over in-memory data structures like HashMap, where you a pay performance price once you exceed your initial capacity.

The database also has a cursor. The cursor is used to read through the database entries sequentially to aid the process of self checking. Ideally, deciding which file to check should be random. However, the JE Edition of the Berkeley DB does not support access through record number. Therefore, the only way to randomly pick a database entry would be to randomly step through the database by a random amount.

Limitations

This component of the system has very few limitations. There is a small amount of state for the database environment, cursor and any entries that are in the cache. The rest of the database is stored on disk. Since SleepyCat was designed to handle large databases, accessing the data does not add significantly to the processing time.

Physical Storage

The physical storage is a dedicated directory on the machine that is running the Disk. All blocks are stored in the same directory.

Currently the prototype, through the configuration of the java.policy file, lets any Java program read or write to that directory. In a live implementation, the java.policy file would be modified so that only signed code would have access to the directory. Then, the files in physical storage would be as secure as the rest of the file system.

The main vulnerability of the physical storage is that one of the blocks stored on it may fail. There are several ways that this can happen. One way, is that a block might have an error in a critical area, making the block unreadable. Another way is that bytes in the file may become damaged, mutating the block's content. Lastly the file system may have some larger scales issues making the directory unreadable. The first can be fixed by rewriting the file. The last problem is so severe that it will require human intervention. However, as long as another Disk in the system did not go down, the file would remain readable. The second one however, is much harder to deal with automatically and usually requires human intervention.

ConnectionChecker

Function

The ConnectionChecker periodically checks the connections between the local Disk and the other Disks in the system. To check if a Disk is alive, the ConnectionChecker sends that Disk an AliveRequest. If this process throws a connection error, the ConnectionChecker considers the Disk in need of being restarted. If this does not throw an error, then the ConnectionChecker waits for a result from the Disk before proceeding to the next one. When it notices that a Disk is down, the ConnectionChecker collects from each working Disk the names of the files that they are currently writing to,

starting with itself. It packages all of the file names together and sends this information to the RestarterService for that Disk.

Security

The ConnectionChecker has minimal security issues. It uses JavaRMI to communicate with other Disks and contains an object pointer to the local Disk. An alternative design would be to make the ConnectionChecker an inner class, giving it direct access to the LockManager. However, the small amount of processing time that this saves however is not worth the added security risk.

Limitations

The major resources that this module requires is the state (memory) needed to support the RMI connections to the other Disks. Again, the design could be changed so that the module makes use of other existing local connections, but this would lead to a cumbersome design.

RestarterService

Function

The RestarterService runs in a separate JVM, but on the same machine as the Disk. When the RestarterService receives a request to start a Disk, it first checks to see if it is already trying to restart that Disk. If not, it then checks to make sure that the Disk is actually unreachable. If it can reach the Disk, then the RestarterService disregards the request. If it can not, the ConnectionChecker restarts the Disk, passing along the filenames sent to it from a remote ConnectionChecker.

Like the ConnectionChecker, this component has minimal security issues. It is a completely remote object that uses protocols (request and responses) that are a part of its package. The main security concern comes if the protocol is abused to create increased traffic on the machine. If code signing were implemented however, then this would help reduce this risk.

Limitations

The RestarterService requires very few resources and can handle high demand. It keeps only a small amount of state, and its traffic is limited by the number of Disks in the System.

Client

Function

The Client is the system component that initiates the reading and writing to a file. When initiating a read/write request the Client first attempts to acquire a lock. LockRequests include the client id, the type of lock to obtain, and the name of the file. The Client sends the LockRequest to a Disk (preferably the one that is running on the same machine) to obtain the appropriate lock. If the lock acquisition is successful, the Disk returns a ticket to the Client. If the acquisitions fails (the ticket will be null), the Client is free to attempt to acquire the lock again. The ticket is then used to create the appropriate request: read or write. WriteRequests include the file name, the ticket, the blocks to be written to the Disk, and the total size of the file. ReadRequests include only the filename and the ticket. If the ticket provided with a request does not match the locking information in the LockManager, then the Read/WriteRequest will fail, and an

appropriate response will be sent to the client indicating the failure. Once a ticket is used that same ticket cannot be used again.

Security

The RAID System client and Disk, as well as the Disks and LockManagers, etc. communicate through RMI. Although not implemented in the prototype, RMI has the ability to use SSL. To configure RMI to use SSL, one needs to create custom RMISeverSocketFactory, RMIClientSocketFactor and secure Server/ClientSockets along with the associated Input/OutputStreams. The SocketFactorys will allow the client and server side (Disk) to use secure communication. Since the entire file (in pieces) is transmitted from the Client to the Disks, it provides increased security for this area of the system.

Limitations

Like the Disk, one of the main strains on this component is memory. The amount of state the Client maintains depends on the number of concurrent accesses. Currently when the Client is preparing to send a file to the Disks, it first reads the file into memory "numberOfDisks -1" blocks at a time (if the file is not already fully in memory). Next, the Client calculates the parity block for that row and begins constructing the WriteRequest that it will send to each Disk. It continues, until it has completely read through the file. Then it sends the appropriate WriteRequest to each Disk. Although this cuts down on network traffic it consumes a significant portion of available memory.

Another option would be to calculate the parity block as it read in the bytes, sending out blocks as it read them in. This would decrease the amount of memory use, but would greatly increase the traffic. The Client is memory intensive when receiving a response from a ReadRequest. The Client stores the entire file in memory, while checking through the data for missing blocks. One solution to this would be to store the blocks to disk as they are received-- then checking the blocks on disk before they are

28

returned. However this would slow down the Client because it would require the Client to make an increased number of calls to disk.

Recovery

When a Disk goes down, it is the job of the ConnectionChecker to detect it and begin the recovery process. If the LockManager or the FileChecker encounter a problem contacting a Disk, they notify the ConnectionChecker of this problem. The ConnectionChecker collects the names of the files that are being written to. The act of collecting these file names, sends the Disks into a read only mode. During this time the Disk will not process any WriteRequests. The downed Disk will be started up again, in recovery mode, by the RestarterService. During this period, the Disk will cycle through each file name passed to it by the RestarterService and begin checking those files to make sure that it has the correct data. *The newly Restarted Disk will not accept lock, read or write requests.* After this processes ends, the recovered Disk will send the other Disks a recovery message indicating that it is ok for the Disks to resume receiving ReadRequests/WriteRequests.

There are several different recovery scenarios that the system would have to deal with. The simple case is that a Disk goes down, and the ConnectionChecker is able to collect all of the file names that currently have write locks. In this situation, the system continues to respond to ReadRequest only. Recovery time is determined by two main components. One is the amount of read traffic remaining on the system. The other is the amount of time it takes to check all of the files that had write locks. The amount of time that checking takes would be closely tied to the amount of write traffic at any given point.

Another scenario is that some write locks are released before the ConnectionChecker can collect the file name. When the Disk is brought online again, it will not check the missed file. If the new information was not written to the Disk before it went down, then file will contain bad data. This information will remain incorrect until the file is written over or the FileChecker corrects it. If another Disk has a failure associated with the same file, then the corrupted version could corrupt the entire file.

In another scenario, the same Disk goes down again, before it has fully recovered. This does not create a significant problem since the ConnectionCheckers of the online Disks continue to function while the system is in recovery. Similarly, once the Disk is sent into read-only mode, the set of files it was writing to is saved, even though those writes may complete. The file names are only discarded once the Disk has fully recovered. Since the system rejects WriteRequest while the system is recovering, no new files can be added to that set. If a Disk goes down again before it can fully recovery, it merely restarts the recovery process.

The most problematic situation is when an additional Disk goes down while another Disk is still recovering or two Disks go down at once. Like the above situation, the ConnectionChecker continues to work when a Disk goes down. Since the WriteSet is not cleared from the Disk until the recovered Disk notifies it that it is online, both Disks will check the same files. When they attempt to check the files however, it becomes impossible to verify the contents of the file. RAID parity encoding by design is only resilient to a single failure. Multiple failures make it impossible to reconstruct the data accurately. For this reason multiple Disk failures shuts the entire system down. An alternative design would be to allow the Disks to service ReadRequest, but to also give the user some sort of warning about the validity of the content. Because of the potentially critical information in the files however, this should not be done with out fully weighing the risks and benefits. Again, the recover time will be determined by the amount of remaining read traffic and the amount of write traffic at any give moment. Like the situation above, there is the possibility of file corruption.

Combined System

Function

The current PING implementation uses the file system for storage. Users of the PING system access and modify records by using PING access methods. For the purpose of this prototype, the PING system was modified to use the RAID system as storage.

This modification required the PING System to implement the typical client for the RAID system. A RAID System client needs only to have a connection to each Disk, a client id and the memory required to calculate parity blocks.

Security

A major concern of the combined system is security of the communication between components. The PING Server has the ability to communicate with the user through SSL. This provides security by providing message privacy and message integrity[11]. In classic SSL communication, when the client and server begin communication, the client uses the server's public key to transmit a session key to the server. This session key is discarded at the end of the session. To ensure message integrity, all messages are transmitted with some sort of checksum to verify that the message content has not been altered in route. As stated before, SSL communication can also be used to help increase the security of communication between RAID components.

Limitations

There are two main areas for possible bottleneck. One is the server that hosts the PING Server. The number of users that are able to connect to the PING server will dictate how many Read/WriteRequests are passed on to the RAID system. The other is the throughput of the RAID System. Since applications servers are a major area of industry development, any top of the line application server chosen would be adequate for a live implementation of the system. The real bottleneck will be that of the RAID System.

Test Setup

A variety of tests were run on the Distributed RAID system. These tests were designed to show system performance under a variety of cases. Table 1 shows the

general statistics of each machine used in these tests. The computers were interconnected by 100Mb/s Ethernet interfaces.

Host name	Processor	RAM (MB)	Disk space (GB)
agent	1000 MHz AMD Athlon	288	8.5
sandiego	800 MHz Pentium III	513	36.4
singapore	800 MHz Pentium III	512	37
merlion	2.8 GHz Pentium 4	897	32.8

Table 1: General statistics of the computers used in testing obtained from examining each computers /proc directory[i].

First, general statistics were collected for each machine. These included benchmarking test and network speed. To determine network speed, each computer in the system sent a ping message to all other computers in the system to determine the roundtrip time between the computers. The message sizes ranged from 56 bytes to 10 kilobytes. To determine the I/O capabilities of the system several benchmarking test were done. For each test, 1,000 files of equal size were written to disk and the average time for each was recorded. The size of the file varied from 1Kb to 56Kb--this test was repeated on each computer. The files were written through a Java program, so that the speeds of the I/O operations could be effectively compared to the Distributed RAID system and PING--both written in Java. Because write operations tend to be the most expensive, performance wise, read tests were not performed.

After information was collected on each of the test machines, the Distributed RAID System was tested. The size of the blocks used in the first test was 100 bytes. First the average read and write times were measured for a variety of different file sizes. Files ranging in size from 100 bytes to 1MB were written to and read from the Distributed RAID System 100 times each. Specifically, the file sizes used were: .1KB, 1KB, 10KB, 43KB and 1MB. 43KB file size was chosen as that is the current best estimate on the average size of a PING record. 10KB and 1M file sizes were chosen as

---

[i] The proc directory contains what the kernel believes to systems feature. Features like the RAM may actually by larger than the kernel believes it to be.

they are probable lower and upper bounds on PING record sizes. Finally, .1KB and 1KB files sizes were chosen to expose any possible trends in file size and file transfer time.

Next, a series of read tests were performed with the FileChecker disabled. In these tests, the portion of blocks missing for the file being read was increased from 0 to **1/d** (where **d** is the number of disks in the system). Next, both the read/write test and the read test with missing blocks were repeated, this time with the number of disks increasing from 3 to 4. Finally, the block size in the Distributed RAID System was increased from 100 bytes to 1Kb to 10kb, and all tests were repeated.

Results

Networking and General Statistics

  For each computer used in the test network, statistics were collected. These include average ping times (out of 1000) for various packet sizes. Table 2, Table 3 and Table 4 show the average ping times.

Host	Packet Size(bytes)	Min(ms)	Average(ms)	Max(ms)
sandiego	56	0.128	0.142	0.286
sandiego	100	0.145	0.158	0.346
sandiego	1024	0.467	0.478	0.513
merlion	56	0.096	0.161	0.286
merlion	100	0.1	0.165	0.228
merlion	1024	0.289	0.354	0.618
singapore	56	0.123	0.14	0.293
singapore	100	0.139	0.152	0.333
singapore	1024	0.461	0.472	0.49

Table 2: Ping statistics for host agent.

Host	Packet Size(bytes)	Min(ms)	Average(ms)	Max(ms)
agent	56	0.123	0.137	0.315
agent	100	0.139	0.151	0.179
agent	1024	0.456	0.471	0.606
merlion	56	0.094	0.158	0.322
merlion	100	0.104	0.168	0.284
merlion	1024	0.301	0.363	0.429
singapore	56	0.128	0.142	0.323
singapore	100	0.142	0.158	0.181
singapore	1024	0.479	0.496	0.655

Table 3: Ping statistics for host sandiego.

Host	Packet Size(bytes)	Min(ms)	Average(ms)	Max(ms)
agent	56	0.1	0.1	0.3
agent	100	0.1	0.1	0.1
agent	1024	0.4	0.4	0.5
merlion	56	0.1	0.1	0.3
merlion	100	0.1	0.1	0.2
merlion	1024	0.3	0.3	0.8
sandiego	56	0.1	0.1	0.2
sandiego	100	0.1	0.1	0.2
sandiego	1024	0.5	0.5	0.7

Table 4: Ping statistics for host merlion.

Next, random files of varying sizes were written to each host through a Java program. The purpose of these tests was to determine how long Java takes to write information out to disk. Table 5, Table 6, Table 7 and Table 8 show this information.

Host Name	agent			
File Size	1KB	10KB	43KB	1MB
Average Time(ms)	0.082	0.136	0.586	20.536

Table 5: Average Java write time for host agent.

Host Name	singapore			
File Size	1KB	10KB	43KB	1MB
Average Time(ms)	0.082	0.135	0.509	20.533

Table 6: Average Java write time for host singapore.

Host Name	sandiego			
File Size	1KB	10KB	43KB	1MB
Average Time(ms)	0.0944	0.167	0.622	25.032

Table 7: Average Java write time for host sandiego.

Host Name	merlion			
File Size	1KB	10KB	43KB	1MB
Average Time(ms)	0.041	0.081	0.251	7.305

Table 8: Average Java write time for host merlion.

Write Benchmark

For the write benchmark, test files with the following sizes where written to the Distributed RAID System 100 times: 1KB with .1KB block size, 10KB with .1KB block size, 10KB with 1KB block size, 43KB with 1KB block size, 43KB with 10KB block size, 1MB with 10KB block size and 1MB with 100KB block size. Each write was timed and the average time was computed. Next, a rough transfer rate was calculated using the following formula:

$$\frac{fileSize + fileSize \bullet \left(\dfrac{1}{numberDisks - 1} \right)}{averageTime}$$

The number of Disks in this set of test ranged from 3 to 4. The 3 Disk Distributed RAID System used hosts singapore, agent and sandiego. In these tests singapore was also the machine that ran the client. Table 9 and Table 10 show the average write times while Figure 5 and Figure 6 show the transfer rates for various block sizes.

File Size	1KB	10KB	10KB	43KB	43KB	1MB	1MB
Block Size	0.1KB	0.1KB	1KB	1KB	10KB	10KB	100KB
Average Time(ms)	135.5	1400.6	1563.8	1263	956.45	13177	23281

Table 9: Average Write Time for 3 Disk Distributed RAID.

File Size	1KB	10KB	10KB	43KB	43KB	1MB	1MB
Block Size	0.1KB	0.1KB	1KB	1KB	10KB	10KB	100KB
Average Time(ms)	60.87	1176.9	877.88	2854.3	883.86	19997	18336

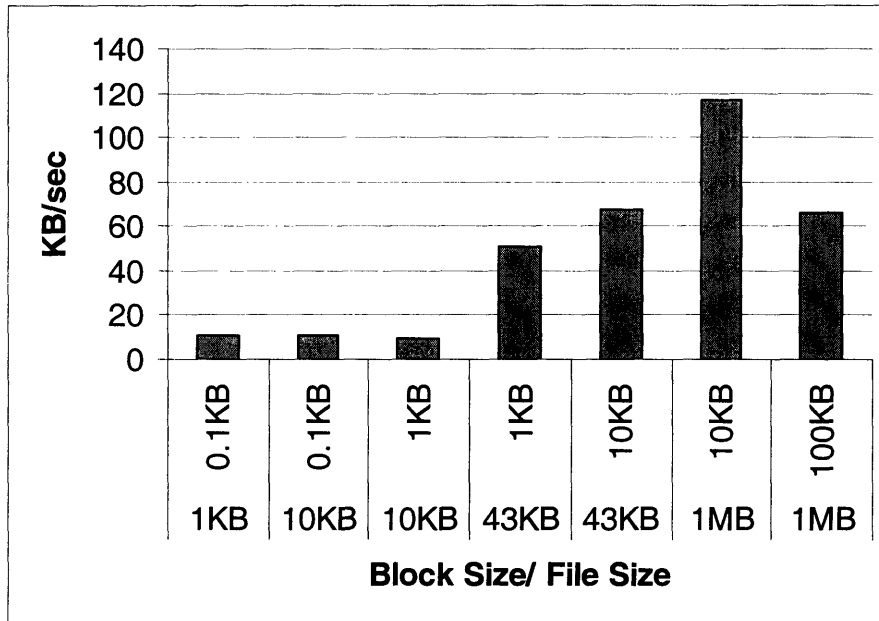**Table 10: Average Write Time for 4 Disk Distributed RAID.**

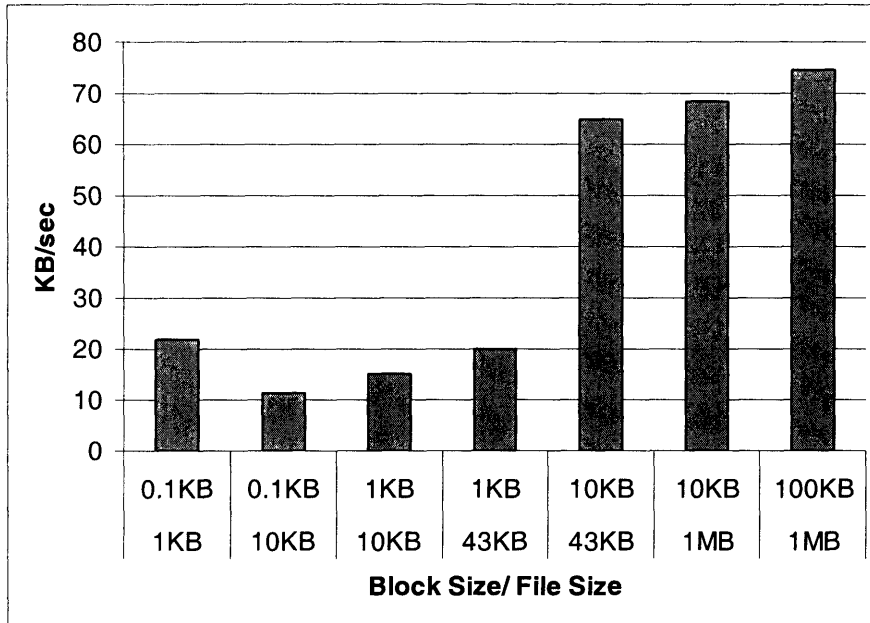**Figure 5: Write transfer rate for 3 Disk system.**

Figure 6: Write transfer rate for 4 Disk system.

Read Benchmark

The read benchmark tests were conducted in the same manner as the write benchmark test. Files with the following sizes were read from the Distributed RAID System 100 times: 1KB with .1KB block size, 10KB with .1KB block size, 10KB with 1KB block size, 43KB with 1KB block size, 43KB with 10KB block size, 1MB with 10KB block size and 1MB with 100KB block size. As before, each write was timed and the average time, along with a transfer rate estimate, were computed. The 3 Disk Distributed RAID System used hosts singapore, agent and sandiego. Similar to the write benchmark tests, for these tests singapore was also the machine that ran the client. Table 11 and Table 12 show the average write times while Figure 7 and Figure 8 show the transfer rates for various block sizes.

File Size	1KB	10KB	10KB	43KB	43KB	1MB	1MB
Block Size	0.1KB	0.1KB	1KB	1KB	10KB	10KB	100KB
Average Time(ms)	58.78	109.11	48.64	102.67	57.5	538.36	473.22

Table 11: Average Read Time for 3 Disk Distributed RAID.

38

File Size	1KB	10KB	10KB	43KB	43KB	1MB	1MB
Block Size	0.1KB	0.1KB	1KB	1KB	10KB	10KB	100KB
Average Time(ms)	68.01	113.26	86.7	103.97	68	516.28	544.93

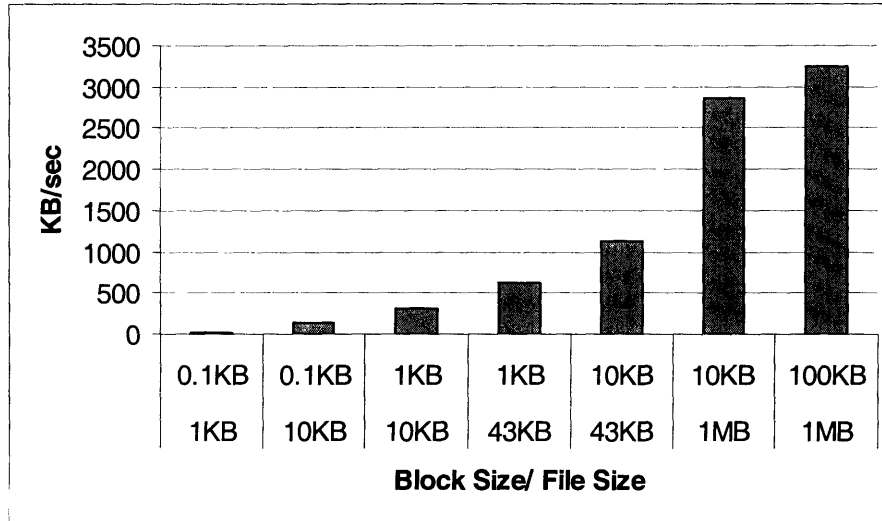**Table 12: Average Read Time for a 4 Disk Distributed RAID.**

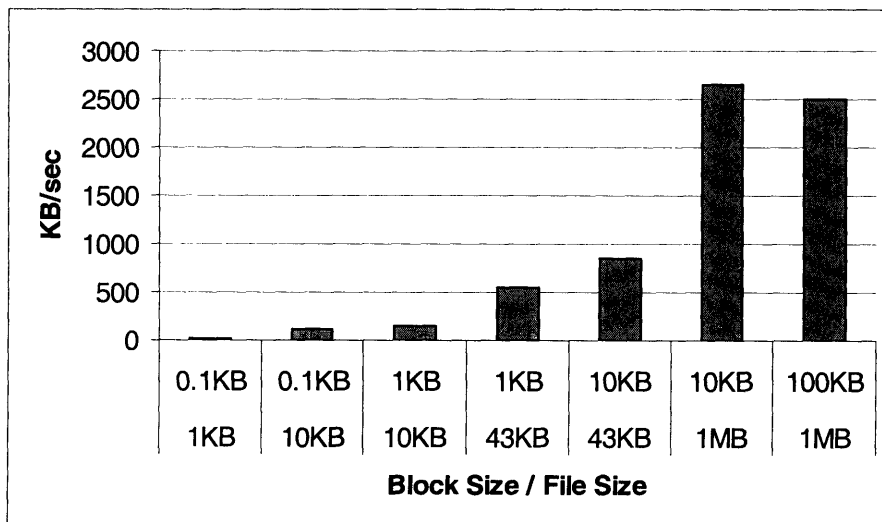**Figure 7: Transfer Read Rate for 3 Disk Distributed RAID.**

Figure 8: Read Transfer Rate for 4 Disk Distributed RAID.

Read Performance with Missing Blocks

The final tests performed looked at the performance of the system when blocks were missing from the file. For these tests 43KB, with 1KB blocks and 10KB blocks, files were used. The tests were repeated for systems with both 3 and 4 disks. Table 13 and Table 14 show the results of these tests. Figure compares the different read transfer rate for a 3 Disk system. Figure 9 and Figure 10 compare the rates for a 4 Disk system.

File Size	43KB	43KB	43KB	43KB	43KB	43KB
BlockSize	1KB	1KB	1KB	10 KB	10 KB	10 KB
Fration Missing	0	0.222222	0.333333	0	0.133333	0.333333
Average Time(ms)	102.67	420.21	491.36	57.5	610.7	728.88

Table 13: Average read time for 3 Disk system with varying amount of blocks missing.

File Size	43KB	43KB	43KB	43KB	43KB	43KB
BlockSize	1KB	1KB	1KB	10 KB	10 KB	10 KB
Fration Missing	0	0.166667	0.25	0	0.142857	0.25
Average Time(ms)	103.97	467.87	319.21	68	391.69	363.57

Table 14: Average read time for 4 Disk system with varying amount of blocks missing.

Figure 9: Average read transfer rate for a 3 Disk system with decreasing fraction of blocks present. The 1KB block reads have an increasing fraction of blocks missing from 0 to .22 to .33. The 10KB block reads have 0 to .13 to .33 fraction of the blocks missing.
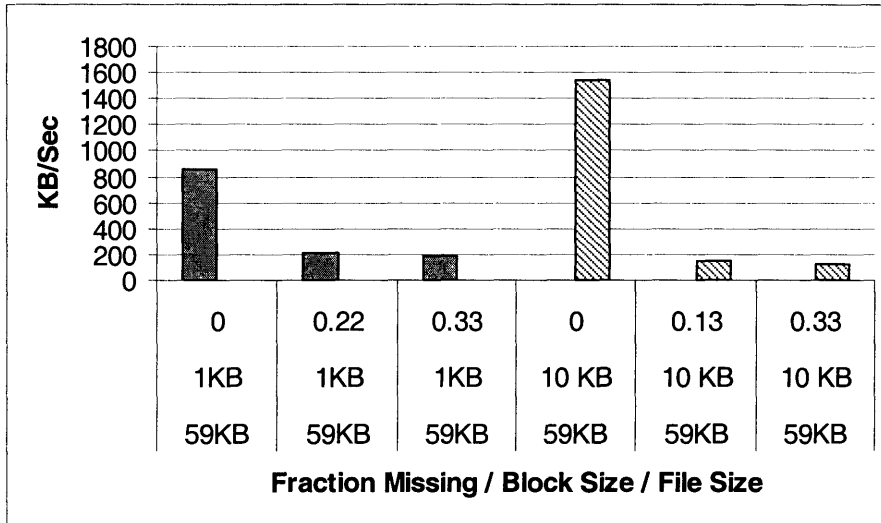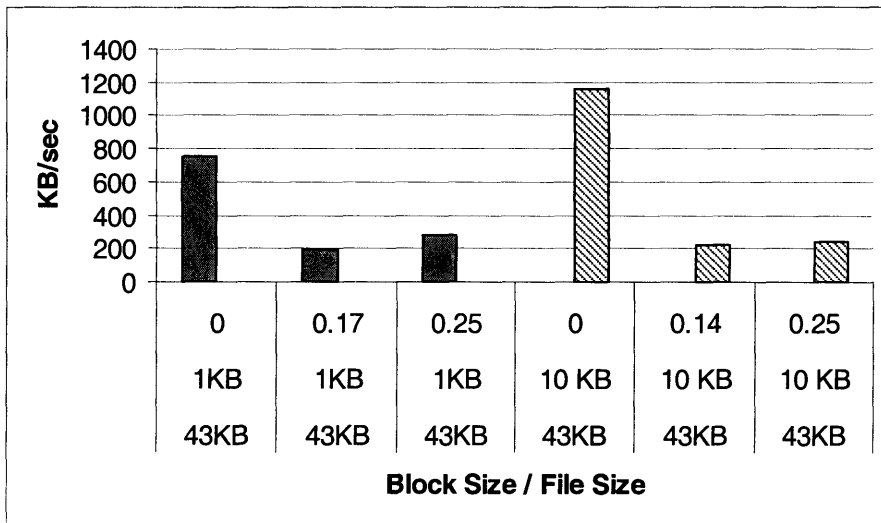

Figure 10: Average read transfer rate for a 4 Disk system with decreasing fraction of blocks present. The 1KB block reads have an increasing fraction of blocks missing from 0 to .17 to .325. The 10KB block reads have 0 to .14 to .25 fraction of the blocks missing.

41

Discussion

Networking and General Statistics

The results of the ping test show that some of the hosts used are closer than others. Specifically, the results suggest that merlion may be slightly father away from the other hosts in the test. This difference is so small however, that it is negligible. Similarly the timed Java test revealed that all the machines used were virtually the same in write speed to disk, with merlion writing files to disk slightly faster.

Examining the minimum average time for each packet size in Table 2, Table 3 and Table 4, we observe that the minimum time for packet sizes 56bytes 100bytes and 1024bytes respectively is .1 ms, .1 ms and .3 ms. Translating that into Mbps we come up with what should be the upper bounds on communication between disks in the Distributed RAID system: 4.27, 7.63 and 26.0 Mbps.

Write Benchmark

The results of the write benchmark tests show that, in general, the write performance of the Distributed RAID System is low for both 3 and 4 Disks. Many of the results place the system at between antiquated 56Kbps modems and current 1Mbps cable modems. The system seems to perform best when a relatively small number of blocks is written. For example, in both the 3 Disk and the 4 Disk systems, the 43KB file written with 10KB blocks outperformed the 43KB written with 1KB blocks. One reason writes with larger blocks might outperform writes with smaller blocks is that they require fewer in-memory data structures to be built. Another related reason that writes with larger blocks perform better may be that they create fewer objects for the Disk to write to physical storage. This does not however account for the system performance for 1MB writes. For the 3 Disk system, writing with smaller blocks worked much better than writing with larger ones. In the case of the 4 Disk system, the performance was about the same. In general the system does not perform well under the strains of encoding a 1MB

42

file. Further investigation is needed to determine if this is an inherent problem with the underlying system, if it is a symptom of poor system memory configuration or some other unknown cause. The 4 Disk system has very similar performance for both the 10KB and 100KB block writes, when writing a 1 MB file. The reason that for 1MB files with 10KB blocks the 3 Disk system outperformed the 4 Disk one is that (by design) it was able to send more information to a fewer number of Disks.

Read Benchmark

The results for the read benchmarks for the Distributed RAID System are fairly good. Both the 3 Disk system and the 4 Disk system wrote large files (43KB, 1MB) at speeds of .5 and 7.8 Mbps. Similar to the write benchmark tests, the 3 Disk system performed better the 4 Disk one. The difference in performance between the two systems is much smaller in the read benchmark case than in the write benchmark.

Read Performance with Missing Blocks

For both the 3 Disk system and the 4 Disk system, an increase in fraction of blocks lost corresponded with a decrease in read transfer rate. Both systems also performed about the same when the fraction of blocks lost doubled. These values shed some light on how reads might perform during system recovery.

Conclusions

Networking and General Statistics

The networking test performed show that the network functioned as expected. Although it would take additional testing to rule it out, it is unlikely that the network had any negative effect on the system.

Write Benchmark

The write benchmark tests performance suggest that the Distributed RAID system might not be the ideal distributed storage for an electronic medical system. The Distributed RAID system write performance turns a high speed (100Mbps) connection into a standard cable modem. Although this would not be a problem if Distributed RAID Clients were run from personal computers, it is a problem since the intended Distributed RAID Client is a hospital system. It is unlikely that hospitals would be willing to participate in a PING System if it takes such a long time to upload files into the distributed storage.

Since the Java benchmarks show fairly quick write times and the ping tests show fast connections, the calculation of the parity blocks is most likely the bottleneck of the system. It is most likely the amount of parity calculations that are slowing the system down. One solution to this problem would be to try the system out on different hardware. Parity calculations are common enough that one can purchase special hardware that makes the XOR calculation (or the encoding scheme of choice) faster. An alternative (and less expensive) solution is to change the method of XOR calculation through code. Highly optimized C code could be called from JAVA in order to compute the encoding more efficiently. One way to get a lower bound on the benefits of this solution would be to replace the encoding algorithm in the Client code with something that returns a byte array of zeros. Making this modification to the Client code and writing out a 43 KB file with 1KB blocks to a 3 Disk system shows a 14 KB/sec increase in performance.

Read Benchmark

The read benchmark results suggest that for retrieval purposes, Distributed RAID System makes a suitable choice for distributed storage of an electric medical record system. However, as previously stated, the poor write performance of the system reduces its overall usefulness. A storage system that easily delivers information but is slow to receive it would encourage users to download information, but never update it. This goes against the information sharing that is at the heart of PING.

Read Performance with Missing Blocks

The same factors that account for the slow write performance of the system are also most likely responsible for slowing down these reads. However, since the maximum fraction of missing blocks for a particular file is $\frac{1}{d-1}$, (where d is the number of disks) reads will never be as slow as writes.

Future Work

There are many opportunities for future research of the Distributed RAID system. One of the most critical areas for future research would be to repeat the tests performed with different hardware and more test machines. As stated previously, the machines used in these tests had to perform parity calculations in JAVA code. It is possible that the different hardware could help improve the write performance of the system. This will help isolate which factors affect the performance of the system, specifically write performance. If the write performance can be improved, then the Distributed RAID System will suitable for large scale distributed storage.

There is also more work to be done on improving the reliability of the system. The current system is only resilient against one fault per stripe. One way to increase the resiliency of Distributed RAID is to use a RAID 6 storage scheme instead of RAID 5. RAID level 6 uses Reed-Solomon codes to increase the amount of redundant information in the system. Although RAID 6 decreases the amount of total data storage (because of an increase in the amount of parity information stored) it does offer protection against two simultaneous failures (two down disks or two faults per stripe).

Additionally, other methods of distributed storage should also be considered. Distributed hash tables (DHTs) is another method of distributed storage that is very similar to Distributed RAID. A distributed hash table is a system in which multiple nodes (computers) communicate with each other in order to coordinate the storage and serial access to files. One similarity between a DHT and Distributed RAID is that both systems lack a central controller. Nodes/Disks in both systems share the responsibility of storing, maintaining and repairing files. While these operations require communication between all Disks in the system for Distributed RAID, a DHT only uses a subset of the nodes in a particular system to maintain a file. Similarly, writing a file to Distributed RAID means contacting every Disk in the system. DHTs use only one node. DHTs can also handle a larger amount of node failures compared to the single failure that Distributed RAID can handle. Two DHTs, OceanStore and Community File System, and their benefits are discussed below.

OceanStore

Overview

OceanStore is a distributed hash table that provides persistent distributed data storage. It allows object location in a severely partitioned network through the utilization of decentralized object location and routing. OceanStore's ability to maintain itself through introspection helps to ensure object locatability. Finally, it uses Byzantine agreement and versioning to provide object consistency.

OceanStore is designed around the persistent object (with floating replicas) and a network of nodes, both with global-unique-identifiers (GUID) from the same namespace. To obtain the GUID of an object, its name is securely hashed (SHA1). The GUID of a node comes from a secure hash of its IP address. In this way, nodes that are neighbors in IP address are not neighbors in OceanStore.

Routing

Every node in OceanStore has a neighbor table which serves as a way to organize information about the nodes that it is connected to. Each node GUID is broken down into four bit chunks (nibbles). Each level in the neighbor table contains the 16 closest neighbors whose GUID match a portion of the node X's GUID. So, the Nth level neighbor-link points to the 16 closest neighbors whose node-IDs match the lowest N-1 nibbles of Node X's id and have different combinations of the Nth nibble. In building the neighbor table, if a match cannot be located, the closest match is accepted.

When a node joins OceanStore, it uses a surrogate node, the node with the closest GUID, to help it learn about its surroundings. This new node uses the surrogate node's neighbor table to help it construct a neighbor table of its own. When a node voluntarily leaves OceanStore, it follows all of its back pointers and informs the nodes that point to it. It also suggests a replacement node to these nodes. When a node involuntarily leaves OceanStore, nodes that link to it fix their own tables. They find a new node to replace the missing value in their neighbor table, but do not inform other nodes. Instead they let other nodes fix their own state[12]. This cuts down on excessive messaging.

Object Location

An object in OceanStore is assigned to a root node. This is the node that has a GUID most similar to the object. When a node decides to store a copy of an object, it sends a message to the object's root conveying this. Along the way, it deposits copies of the object. This can be seen below in Figure 11.
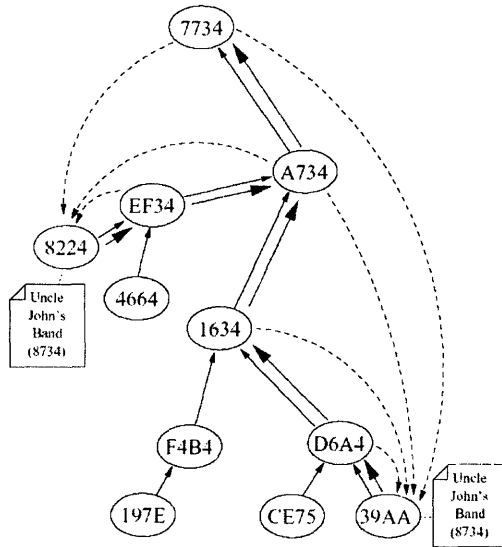

Figure 11: Object Publication in OceanStore[13].

OceanStore uses two methods of object location. One is a fast probabilistic method and the other is a slower, deterministic method. When attempting to locate an object, OceanStore first attempts to use the fast probabilistic method. This involves each node checking its Bloom filter.

Bloom filters are a way to index information. They use an array of width **w** and **n** hashing functions. The output of the hashing function is a number from 0 to **w-1**. To represent having an object, the object name is hashed **n** times with each hashing function. The output of the hashing functions determines which bits in the array to set to 1. Figure 12 below shows an example of a Bloom filter that represents the indexing of two objects.
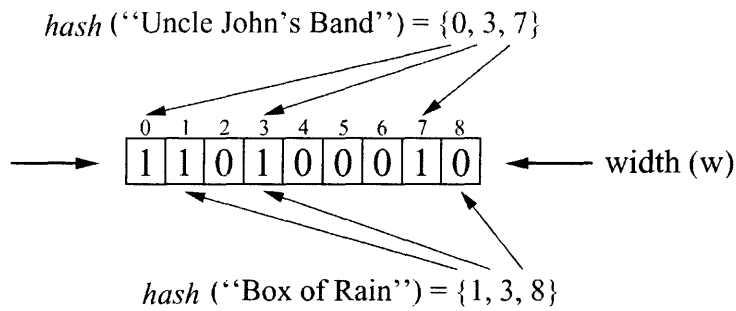
hash ("Uncle John's Band") = {0, 3, 7}

$$\boxed{\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}} \longleftarrow \text{width (w)}$$

hash ("Box of Rain") = {1, 3, 8}

Figure 12: Example of a Bloom filter with 3 hashing functions and width 9^{13}.

A node in OceanStore stores a multi-level Bloom filter for its 3 closest neighbors in latency. It uses these attenuated Bloom filters to reference the information that their neighbors store. For a given outgoing link, each level in the attenuated Bloom filter represents objects that are reachable from an increasing distance starting from that outgoing link. Figure 3 shows an attenuated Bloom filter for the outgoing link from node A to node B. Because Bloom filters can give false positives, each level in the attenuated Bloom filter has a decreasing probability associated with it. In Figure 13, the probability that going through node B to node A will find "Uncle John's Band" is $1/4 + 1/8 = 3/8$. After **d** (the depth of the attenuated Bloom filter) hops, the probabilistic method gives up and hands the query over to the deterministic method.
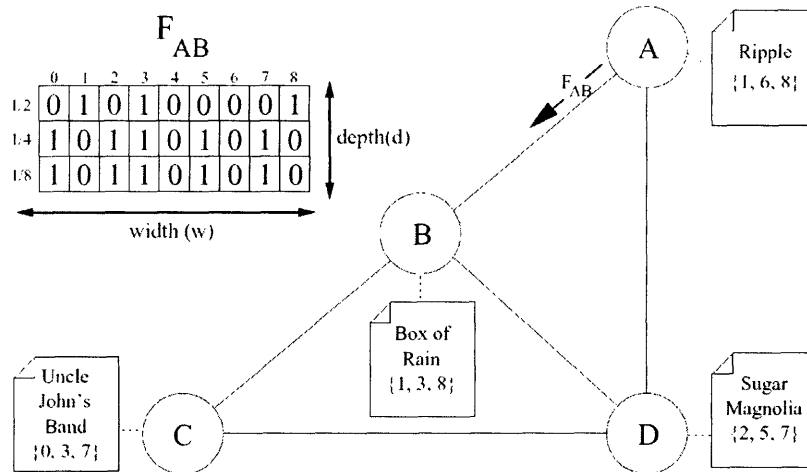
$$F_{AB}$$

Figure 13: Attenuated Bloom filter and corresponding nodes[13].

To locate an object using the deterministic method, a node routes a message toward the object's root until it encounters either a cache pointer or the object root node. For example, referring again to Figure 11, if node 197E wanted to locate the object "Uncle John's Band" (8734) it would send a message toward the object's root, Node 7734. However, once has it reached node 1634, it would find a pointer to a storage location.

Updating

Every object has an inner ring: a group of nodes that are responsible for serializing object updates and keeping the current version of an object[14]. Object updates are initialized by clients and routed toward the inner ring. The nodes that make up the inner ring use a Byzantine agreement protocol to agree on a new version. They then sign this new version and give it a timestamp.

Combining with PING

OceanStore has the ability to work well with PING. A single node in OceanStore would have the ability to act as the interface between it and PING. Also, since OceanStore is designed to handle multiple clients, the PING Server would not need to be

50

centralized. Many PING Servers could be placed around the country in order to better service record demand. The downside of using OceanStore as storage is that it requires public and private keys for each file in the system. There are two ways to address this problem. One is to use one public key/private key pair for the entire system. The drawback (also present in the Distributed RAID system) would be that if this pair is discovered, all files in the system become readable. The second is that each patient could entrust the PING Server to store their public and private key.

Cooperative File System

Overview

Cooperative File System (CFS) was developed by CSAIL's Parallel and Distributed Operating Systems group. Like OceanStore, this DHT seeks to create a system of nodes that provides load-balanced data storage and guarantees object locatability. Figure 14 shows the CFS software file structure.
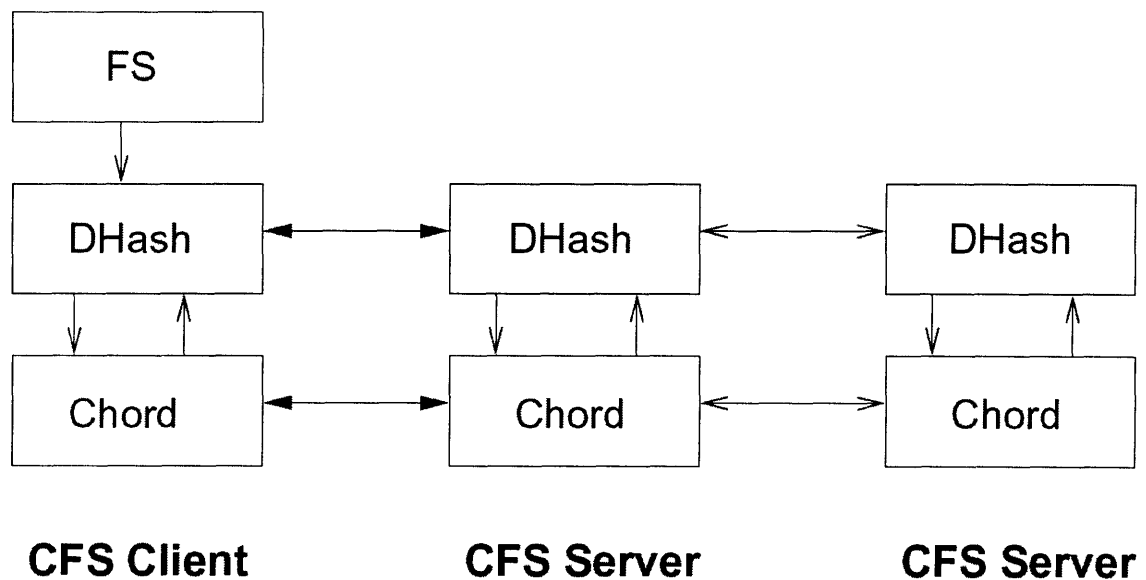
Figure 14: Cooperative File System software structure[15].

CFS has 3 layers: FS, DHash and Chord. The FS layer interprets blocks as files and presents a read-only file system interface to applications. The DHash layer stores data blocks and the Chord layer maintains routing tables used to find blocks15. Similar to OceanStore, a CFS node is named by the secure hash of its IP address and virtual node index. Again, this is to relax the system dependencies on IP structure. Files can be modified only by their owner, and only through complete replacement. CFS does not allow a delete: objects that are not renewed will simply expire.

Routing

Every node in CFS keeps a successor list and a finger table. A successor of an ID **j** is the node with the smallest ID that is greater than or equal to **j** (with wrap around)[15]. Each node keeps a list of identities and IP addresses of its **r** immediate successors on the Chord ring (based on the naming scheme mentioned above). This allows Chord to implement incremental routing that, like the post-fix routing of OceanStore, is the slower method of object location. When a node **n** joins the system, it asks an existing node for **n**'s successor[15]. It then asks its successor **m** for **m**'s **r** successors (**m**'s successor list). To reduce the number of hops necessary to locate an object using incremental routing, CFS nodes also store a finger table. This finger table stores the identities of nodes at power of 2 distances away from it.

Object Location:

Like OceanStore, object names are hashed and given IDs with the same namespace as the node IDs. They are then assigned to the node with the ID that most closely matches its ID. Block replicas of the object are placed on successor nodes in the Chord ring. Again, because these nodes in ID are not nodes in IP, the replicas are fairly independent. To locate an object, a node first tries to use its finger table (the faster of the two methods) and then uses its successor list. In addition, CFS implements object caching along the query path.

Combining with PING

High reliability and the ability to support multiple node failures also make CFS a good candidate for the distributed storage of PING. However there is a major problem that might make CFS difficult to configure as storage for PING. CFS requires that information be updated periodically in order to remain in storage. It is possible for files that have not been updated for a long time to be non-explicitly deleted. For patients who do not see a doctor regularly, this could mean a silent loss of patients' records.

Mirroring

Although RAID like systems and distributed hash tables provide innovative ways to distribute and manage storage they are not the only distributed storage options. The ability to buy cheap reliable servers with a large amount of storage makes mirroring a feasible option. Mirror has the benefit of ease, it requires so special hardware or complex encoding software. Similarly, the act of copying the information is relatively easy, so I/O requests will not drag down the system.

Mirroring does have the problem of a central point of failure. If the main server goes down all files are available for reads but not for writes. Some mirror systems are designed to allow writes if the main server becomes unavailable. The mirrors then attempt to synchronize their data with main server when it becomes available again. However conflict resolution and file merging cannot always guarantee the desired output. When the files in question are medical records, lost transactions may not be acceptable. It will require further research to determine if the good of mirroring outweighs the potential bad.

[1] Garner Karen http://www.sutterhealth.org/about/news/news_emr.html

[2] http://www.caregroup.org/

[3] Riva A, Mandl KD, Oh DH, Szolovits P, Kohane IS et al., *"Personal Internetworked Notary and Guardian"*, Internaltion Journal of Medical Informatics, 2000.

[4] Mandl KD, Szolovits P, Kohane IS. "Public standards and patient control: how to keep electronic medical records accessible but private", British Medical Journal 2001;322:283-7.

[5] http://ping.chip.org Children's Hospital Informatics Program, 2004.

[6] Chen Peter, Lee Edward, Gibson Garth, et al. *RAID: High-Performance, Reliable Secondary Storage*. ACM Computing Surveys, Vol.26, No 2. June 1994

[7] Saltzer Jerome H., Kaashoek M.Frans. Topics in the Engineering of Computer Systems, Feburary 2003.

[8] Saito Yasushi, Frølund Svend, et al., "FAB: Building distributed enterprise disk arrays from commodity components", *ASPLOS 2004*.

[9] Java RMI Forum http://forum.java.sun.com/thread.jspa?forumID=58&threadID=448547

[10] http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html

[11] http://www.first-american.net/SSL.html

[12] Hildrum Kirsten, Kubiatowicz John D., Rao Satish, and Zhao Ben Y., "Distributed Object Location in a Dynamic Network", Theory of Computing Systems, November 2003.

[13] C. Rhea Sean and John Kubiatowicz. *"Probabilistic Location and Routing"*, Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies *(INFOCOM 2002)*, June 2002.

[14] Rhea Sean, Wells Chris, Eaton Patrick, Geels Dennis, Zhao Ben, Weatherspoon Hakim, and Kubiatowicz John, *"Maintenance-Free Global Data Storage,"* IEEE Internet Computing , Vol 5, No 5, September/October 2001, pp 40-49.

[15] Dabek Frank, Kaashoek M. Frans, et al. "Wide-area cooperative storage with CFS". SOSP, October 2001.