Distributed Storage

Evan Danaher

May 31, 2004

Abstract

In many computer labs, there are large numbers of computers with unused drive space. There may also be relatively large quantities of data to be backed up. The goal of this project was to develop a system for storing data distributed over many computers, with enough redundancy so that data can still be recovered if several of the machines are unavailable (due to inevitable hardware failure). The RSraid scheme was chosen for maximum flexibility, and the result is an extremely robust, though very user unfriendly backup system.

1 Introduction

The problem of effectively backing up critical data has been around for many decades. Traditional backup methods use a tape drive or other device to store data separate from the main computer. However, this requires the purchase of an extra drive and extra media to use for backups, as well as human labor to ensure that tapes are stored safely and not overwritten.

A different method has become possible in the past few years, due to the prevalence of large hard drives with significant unused space in standard desktops and networking of these computers. This unused space can be used for automatically backing up data at no additional expense. However, since desktops are often less reliable, this backup method requires data to be recoverable, even if a significant amount is lost. Redundancy techniques provide this ability.

An article, *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems*, describes the use of a variant of Raid-Solomon for use in certain applications. The idea is that Raid-Solomon can be used relatively simply when you know which data is lost; normally it is used in devices such as CD-ROMS, where it is impossible to know which data is correct and which was mangled. But in "RAID-like Systems," it is known which device failed, so data recovery is simpler. In particular, I can use this algorithm for recovering data from multiple computers when some computers are dead.

1.1 Scope

This project is designed for a fairly limited but still significant group: organizations that have large file servers to store most data, as well as dozens or hundreds of "user" machines that are relatively new. Only these groups will be able to use this program: with no central server to back up, there is no need for it; without a number of "user" machines, there is not enough space to store the backup. However, this category still contains many groups, such as businesses and schools. Even large businesses with existing backup systems may find this

useful; once set up, it requires little maintenance, and can recover backups quickly compared with existing solutions.

It could also be extended for use in organizations that currently don't use a single central server. If each user machine is running both the server and client programs (or more likely, they are combined into one program that handles both functions), the user's data could be transparently backed up onto other user machines. And if performance could be increased, it may even be able to serve as a general network filesystem, though this has not been implemented at all.

This project is explicitly not intended for any sort of distributed backup over the Internet or using any outside computers. As a result, encryption and other security-enhancing features were ignored. Instead, a focus was on distributing computation to make computations more efficient.

2 Background

In order to efficiently use existing computer hardware and safely store data, a system must be devised for storing that data. This project aims to provide such a system.

The basic project would be to choose a method for redundantly storing the data, implement it, and implement a method for distributing and retrieving the data over multiple computers. If there is extra time, the project could be extended to include privacy features, distributed computation of the redundancy, or other useful extras.

2.1 Existing Solutions

Some commercial solutions and at least one free solution are available to perform backups over many other computers. However, these are generally intended for backup to other computers over the Internet. As a result, they assume that all encoding must be done on your own machine: it is relatively slow to send data out over the Internet, and you would not want to send your un-encoded data to someone else's computer. If the encoding and decoding computations can be spread over multiple computers, the backup can be created and read more quickly. This project makes it possible to perform this distributed computation in the case where backups are performed on local computers.

There are other file systems that store data distributed over multiple computers. However, these are generally either hardly redundant at all, or overly redudent. Some provide no data security beyond having RAID on the individual servers; if a server dies, so does that data stored on it. Others, such as the Google File System, go to the other extreme, storing everything on (at least) three different servers. This, while fairly simple to implement, wastes a huge amount of data, and requires constant monitoring: it is possible for the "wrong" three servers to die, still resulting in data loss. GFS avoids this by actively monitoring data, and whenever data exists on only one server, doing a high priority copy to another server. But for a backup system such as the one proposed, this is unrealistic, as well as unnecessary.

3 Theory

3.1 RSraid

Reed-Solomon coding is designed for use in systems where data loss is expected, and it is impossible to tell where the data is lost. However, in this project, it will be known which data was lost (which computer died), and so the full power is not needed. A simpler version, dubbed "RSraid" or "Reed-Solomon for Raid-like devices" is put forward in [1].

The basic idea is to put the data, d_0 through d_{n-1} into an $n \times 1$ matrix, and multiply some other $n + m \times n$ matrix by it, resulting in another $m \times n$ matrix. Each row in this matrix represents the data to be sent to a certain device. As a result of this, if n devices out of the $n+m$ are available, an $n \times n$ matrix can be constructed and the resulting equation can be solved for the original data. There are two tricky parts to this: that multiplying matrices tends to leads to large numbers, and how to find an initial $n + m \times n$ matrix to multiply by so that the data can be recovered from *any* n rows of the final matrix.

The first problem, that of large numbers, is surmounted by using *Galois fields*, also known as *finite fields*. They are most rigorously thought of as polynomials up to a certain degree, taken modulus some prime polynomial. However, then can be efficiently implemented using binary arithmetic, where "addition" and "subtraction" are both done as binary XOR, and multiplication is done using a table of "logarithms". While this is an odd way to treat numbers, the result is a consistent set of rules that can be used for multiplication and addition of numbers less than 2^n with results also in this range the perfect solution for multiplying matrices on binary-based computers.

The second is solved as shown in [1] - using the matrix where $a_{i,j}$ is equivalent to i^j, that is:

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 2^2 & \dots & 2^n \\ 1 & 3 & 3^2 & \dots & 3^n \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & m+n & (m+n)^2 & \dots & (m+n)^n \end{bmatrix}$$

This matrix is calculated in the finite field, giving a set of values all less than 2^n. Then, to speed computation, it is reduced by a Gaussian elimination type so that the top $n \times n$ matrix is the identity matrix; this means that n out of the $n+m$ rows are simply the original data; this has a significant savings in computing time.

3.2 Communication

The other part is to distribute these pieces over various computers - this was be done using standard sockets, though several functions were written to handle buffering of data. The RSraid algorithm is inherently byte based, so individual bytes were frequently written onto the network. In order to avoid some of the overhead associated with small packets, these individual bytes were buffered locally, and only flushed when necessary. Still, networking was a major bottleneck as many computers were attached to the system.

4 Procedure, and Workplan

1. Learn about existing methods for redundant storage. One possibility for this is Reed-Solomon encoding, a common method. Then this should be implemented either using new code, or using an existing library.

2. Once files can be stored redundantly, split the file up among smaller files, such that the original data can be reconstructed for various subsets of the files.

3. Write code to distribute the files onto and retrieve them from different computers.

4. If time is left over, add additional features such as distributed computing for increased encoding and decoding speed and random access to backups.

5 Discussion

The basic RSraid algorithm was not too complicated; it was mainly a matter of learning it and implementing it as written. Distributing the files from RSraid over multiple computers was surprisingly non-trivial: I had issues with knowing when the data was finished. After trying fairly complex methods of sending and end-of-file character, I gave up on these since they ended up being too complicated. The final solution was to send the size of the file first; that way the receiving program knew exactly how much to expect. This is more efficient than and end-of-file sequence: if that sequence appears in the file, it must be escaped to signal that it is not in fact the end of file. It is also significantly easier to implement, though

it does require that the size of the file be known ahead of time. However, if the size is not known, it can be stored locally before being distributed.

5.1 RSraid

RSraid was actually the simplest part of the project - the reference paper was well written, and implementing the algorithm as given was straightforward. I did have to make a slight change when I stumbled upon an errata to the paper, pointing out that the given transformation matrix would not actually work. The original matrix was $T \times n$, with the upper $n \times n$ matrix the identity matrix, and the remainder a Vandermonde matrix. However, this apparently does not work in all situations. The proper matrix, as given in the errata, is obtained by reducing a full $T \times n$ Vandermonde matrix so that the upper portion is the identity matrix. (Of course, this reduction to the identity matrix is not strictly necessary, but speeds computations tremendously, especially when this un-encoded portion of the data can be used for decoding.)

Aside from that, implementing the algorithm was a matter of writing out a few small functions to handle Galois Field arithmetic, and a few more to handle the matrix multiplication using Galois Fields.

5.2 Distribution

Distributing the data over computers, which I thought would be easy, turned out to be fairly difficult. The basic sockets code seemed to work fairly well, but I had some issues with the

details. First, how to know when the file was finished? Initially, I was using an end-of-file character. However, since the (encoded) data could include any character value, this EOF character could appear in the file. Here it had to be escaped, or else the file would end prematurely. Finally, after a few days of trying to get ever more complex schemes to work, I decided to try an alternative.

Another way to know when the EOF has been reached is to begin the transmission with the length of the file. While it has some disadvantages, such as requiring knowing the length of the file ahead of time, it is much simpler, as the same data is always sent every time, and there is no requirement to escape certain characters. This turned out to be successful, and worked fairly well.

However, there was another issue: speed. Even over a 100Mbit network, I was unable to attain speeds over a few KB/sec. I could find no reason for this, and never managed to fix it. I thought the issue might be that data was being sent with only one byte per network packet, resulting in huge amounts of overhead. But when I implemented a buffered sockets setup, so that data was only actually written to sockets every 1500 bytes, there was no speedup. In order to be useful, this problem would certainly have to be fixed, but I have no idea what to do about it.

6 Results, Conclusion, Recommendations

The resulting system, while not completely finished, represents the core of a distributed backup system, and could potentially grow into a full redundant networked filesystem. The

main impediment to real-time access is not, as I initially thought, processing speed, but rather network load. The current system requires each byte of data to be transferred across the network three times. Over a 100Mbit network, this limits total throughput to less than 5MB/s, assuming that is the only network usage. This is slower than many modern hard drives, and slower than encoding/decoding speed. Thus, this system will probably not be practical for normal storage until gigabit Ethernet is commonly available. Even at that point, it will probably be slower than local drives, though not by as much.

An alternative would be to lessen network traffic by doing the computations on the local computer. This removes the major advantage of this project, but would make it more reasonable. Especially if most computers are generally available, in which case most data could be read directly with no decoding required. This would probably form a practical filesystem.

The system is also not at all user friendly - it requires the user to know all the computers data is stored on, while this data should be stored in the backups. Multiple file handing is also not as good as it could be. However, both of these should be fairly easy to add on, given enough time. But until they are added, it will not be useful in general. With them (and proper networking as specific above) this could be a very practical backup solution in some cases.

Overall, this project was a success. The goal was to create a reasonably fast extremely redundant backup system making use of no additional hardware beyond standard user machines, and it succeeded at this goal.

7 References

A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems <http://citeseer.nj.nec.com/plank97tutorial.html> was my main resource for coding the RSraid algorithm.

The Distributed Internet Backup System <http://www.csua.berkeley.edu/ emin/source_code/dibs/index.html> seems very similar to what I'm doing.

Sockets Tutorial <http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html> was my main reference for socket programming (distributing across multiple computers).

Note: Correction to the 1997 Tutorial on Reed-Solomon Coding <http://www.cs.utk.edu/ plank/plank/papers/CS-03-504.html> had the proper encoding matrix for RSraid.

The Google File System <http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf> describes a very different distributed filesystem.