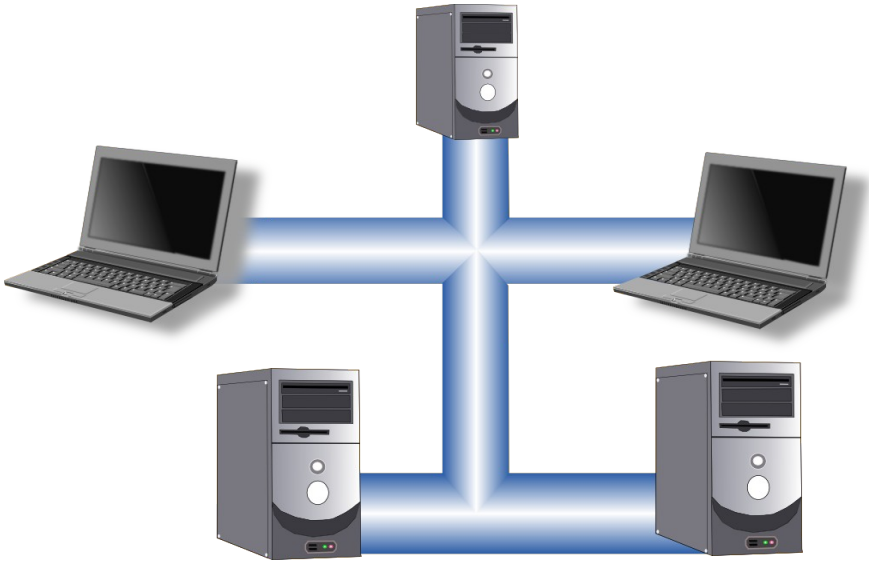
Title

Jason's Distributed Desktop Backup System

By **Jason Long**
A project for COMP 512
Penn State Harrisburg
5 May 2008.



Final Report

Table of Contents

Introduction

For this project I have designed a distributed desktop backup system. It is a backup system targeted for desktops, where the desktops being “backed up” are only sporadically available, each desktop has spare disk space, and there is a lot of redundancy between the backups (i.e. the participating desktops have many of the same files). It is “distributed”, avoiding a central controller, which would be a single point of failure, and a scalability bottleneck; instead, it uses a “peer-to-peer” architecture, where each participant is responsible for a portion of the overall work.

The result is a program that can be installed as a system service on several or many workstations on a single organizational network (it is not designed to run across a slow-speed wide-area-network-style link). The system administrator configures a certain amount of disk space that each workstation is allowed to use for the purpose of backups. Each workstation running the program will then periodically perform a full “backup” of itself. The backup will be broken up into chunks and transferred among the other workstations on the network.

This project is an attempt to build a unique and innovative distributed system, using principles taught in my COMP 512 class. In this case, it is a “distributed” solution to the problem of backups, relying completely on a peer-to-peer network on which files can be replicated. In fact, without the participation of peers, the backup system is completely useless, because there would be no replication, and therefore no way to “recover” in the case of storage failure. My solution is not the only way to implement backups in a distributed system... it is a unique solution in that I'm avoiding the use of a central coordinator.

Related Work

Existing desktop backup solutions can be categorized by where they write backups to. Most common backup destinations are 1) a locally attached device (e.g. a magnetic-tape or optical drive), 2) another system over the network, or 3) an unused portion of the local disk drive. Some products support more than one of these destinations. My solution does not require attaching devices to each desktop, so I will ignore the first category.

Of the programs that backup to another system on the network, they almost always involve a

centralized server. The centralized server may be on your own network, or it may be in a service provider's secure data center. Some desktop backup programs, such as Norton Ghost [1], are “image based”, which emphasize an easy restore of the entire system. The result of the backup is a single image file that can be used to restore the complete system. Other desktop backup programs, such as GoBack [2], are file-based, that backup files as individual units, making it easier to restore or roll-back individual files to earlier versions.

Solutions that backup to an area of the local disk are protecting against human and software errors, rather than hardware failures. (If the local disk fails, the backed-up data becomes unavailable.)

Another set of backup utilities provide some peer-to-peer like functionality. BackupCoop [3], for instance, is a program to let people pair up over the Internet and exchange full backup images with their partners. Another example, BuddyBackup [4], is file-based, and replicates those files with one or more “buddies” over the Internet. I was unable to find any *products* that provide a dynamic, automatic, peer-to-peer backup solution, where the discovery of peers is automatic.

Although I didn't find any products that were automatic peer-to-peer, I did find a paper describing a research project implementing such a system. pStore [5] stores files in a peer-to-peer network, using a distributed system based on Chord (described later), and it appears to use a content-based addressing scheme. This makes it very similar to my proposed system. However, their content-based addressing appears to only cover files; they do not take it as far as I do when I used content-based addressing for directories as well. Furthermore, pStore distributes replicas randomly throughout the ring, instead of to adjacent nodes like my project does. Finally, pStore uses different concepts to deal with garbage-collection and tracking backups.

Problem

Network backups are traditionally done from one or more regular servers to a dedicated backup server, where the backed-up data is written to high-capacity magnetic tape. These magnetic tapes can then be taken out of their drives and stored in an off-site location to provide disaster recovery. This strategy works pretty well when the critical data is on those regular servers.

The problem is that much of an organization's critical data is not just on those file and database servers. This is especially a problem when employees have and use laptop computers so

they can work when they are "offline" (i.e. not connected to the organization network). Therefore, it's desirable to make full backups of every laptop and desktop computer as well as the server. Unfortunately, due to the number of these computers and the sizes of those hard disks, implementing a full desktop/laptop backup scheme is difficult, and expensive.

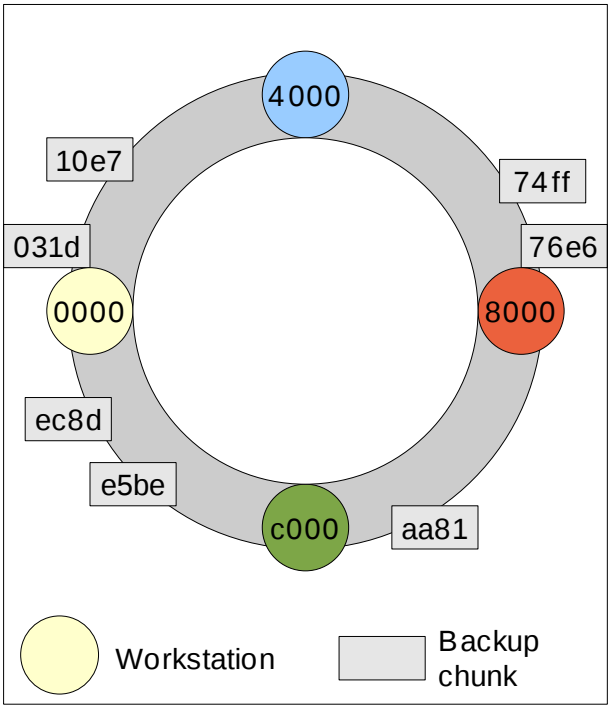
This project relies on the observation that when there is a network with many desktop computers attached, there is 1) significant unutilized disk space, 2) many files duplicated throughout the network, and 3) a legitimate need to perform full system backups of those desktop computers.

Design

The design of this system features three programs that work together to make up the distributed backup system. The three programs are the “server”, the “backup” program, and the “restore” program. The server program runs continuously, in the background, communicating with other computers on the network and exchanging files with them. The backup program is run periodically, either manually invoked or on a schedule; it creates a backup image, splits it into chunks, and uploads those chunks to other workstations on the same network. The restore program reverses the process.

Server Responsibilities

The workstations running the “server” program form a logical “ring” of workstations, each maintaining connections to the “next” and “previous” workstations on the ring. The positions on the ring are randomly picked the first time the program is run. On subsequent runs, it rejoins at the same position it had previously. Backup chunks are also given identifiers, and positioned on this logical ring according to those identifiers. Each workstation is assigned the responsibility of storing the chunks that precede it on the ring. This is basically the same as the Chord project's



distributed hash lookup primitive [6]. In addition, the chunks are replicated to the next several workstations to provide redundancy.

Each workstation only knows its own objects. If a workstation is asked about an object it does not have, it may refer the caller to a different workstation—one that is more likely to have the object. The “next” and “previous” node pointers may be used for this purpose. In addition, as in the Chord project, the workstations hold “finger” pointers to nodes at distant locations on the ring. These pointers allow a workstation to refer the caller to a node much closer to the desired object. This allows the system to quickly find a particular location on the ring, given that each node has only limited information about the ring.

When a workstation wants to join the ring, it first needs to perform “discovery”. Discovery occurs by sending a packet to a certain multicast address/port and listening for replies. All participants listen for these discovery packets on that multicast address/port, and any node that receives such a packet will issue a reply with information about the ring. The joining workstation picks from all the replies the best node to join to, and proceeds with joining to that node.

The algorithms for joining to the ring, and then for maintaining the ring are designed according to the Chord project. Basically, each node will periodically notify its “next” node, and ask it if there's a closer node in between. If so, the node will join in front of the other node. The nodes also periodically make queries to find out what nodes are at remote locations on the ring; these queries are to maintain the “finger” entries.

Network Communication

The software employs three different types of communication. The use of multicast packets has already been mentioned. In addition there are unicast UDP packets and TCP streams. The unicast UDP packets are used to implement the ring joining and ring maintenance algorithms, and “lookup” requests. Lookup requests allow a client or a fellow node to ask where an object should be stored. A lookup request starts with any node, which may give an answer or referral. If a referral is used, the asking node must repeat the question to the node specified in the referral. Eventually, the asking node will ask the node responsible for the object and receive a positive response.

TCP streams are used to transfer actual chunks of the backup. TCP gives a number of benefits, including packet retransmission and reordering, and flow control, which are helpful when transferring actual data. The backup and restore programs establish TCP connections to multiple nodes of the network to perform their respective operations. The server program

establishes TCP connections to the node's immediate neighbors to replicate the contents of its repository to its neighbors.

Replication is simple in concept, but complicated to implement. The goal is to ensure that *n* copies of each backup chunk are in existence at any time, where *n* is big enough to ensure a restore will be possible even when some nodes are unavailable. Implementing is hard because no node can know exactly how many copies actually are in existence. This is discussed later in this paper.

Backup in Chunks

This is how the backup program splits up the backup image. First of all, a backup chunk's *identifier* is a cryptographic hash of that chunk's content, which makes this a “content-addressable” storage system. Content-addressable storage systems are actually easy to implement, and easy to make distributed, because files, once written, cannot change. Files cannot change in this system, because if a file was changed, it would have a new cryptographic hash, and therefore a new name in the system. It would be stored as a new file. In addition, using a content-addressable storage system has the benefit that if two workstations were backing up the same bit of content (very likely to occur, since the machines have similar content), they will both store it at the same single location on the ring.

Now, the way the backup program splits the image into chunks is similar to the “Git” source-code management system [7]. Individual files will be stored as “blobs”, addressed by the hash of their contents. These blobs will be referenced by a “directory” object, which is simple a file listing the names of files and the hashes of the files' contents. These directory objects are also addressed by the hash of their contents. The directory objects are referenced by other directory objects, all the way up to the root of the filesystem. The root of the filesystem then will have a single hash, which can be used to restore the entire system.

Because the backups are split into chunks, the full backup can truly be distributed to many other systems on the network. This would actually make the system less reliable (since a restore is only complete if all chunks are available), but because the chunks are replicated it is very reliable. Also since the backups are in small chunks, a workstation without enough disk space for a whole image can still make a meaningful contribution to the overall system.

One goal is to allow administrators to control how much disk space a particular computer is making available to other systems. This can be achieved by programming the server to have control over the size of the portion of the ring it represents. If a node is positioned in such a way

that it has too big a “piece of the pie” it can change its own node-identifier so that its position has a smaller piece of the pie. However, to truly limit how much disk space a node uses, it needs to be able to handle the situation where it hits its limit. This is not yet part of the design, but it is discussed under Challenges, below.

Security

In order for this system to be usable, some provision must be made for securing the users' backups. In many organizations, one person may have sensitive files on her computer, files which another person should not be able to see. If a distributed desktop backup system was implemented as described here, the second person may actually have copies of those sensitive files on his computer. Without some sort of encryption, the second user may look in the backup program's data directory and find the sensitive file.

Obviously, the solution to this is to encrypt the backups. However, this creates two potential problems. First, the restore program will need the encryption key to restore the file. How do you ensure the first user can restore but not the second user? Second, if two people backup the same file, but encrypt it using different keys, the files will no longer have the same content hash, and therefore will not achieve the space savings described above as an advantage to using content-addressable storage.

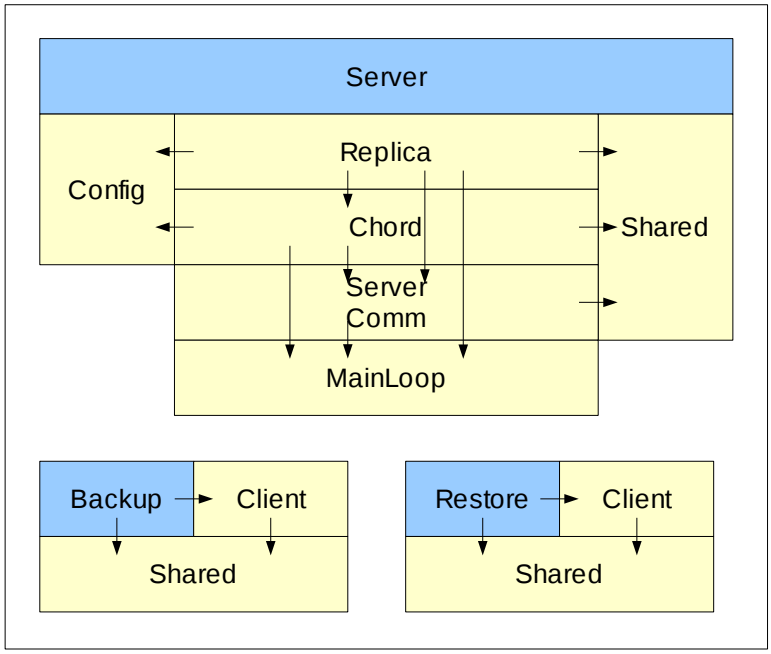
Problem one is solved by pairing the encryption key along with the backup-identifier. So, both in directory tree objects, and in the backup-identifier returned to the user after running “backup”, an encryption key will be specified. Each chunk will potentially use a unique encryption key, but the user only need see the top object's encryption key, since the top object is a directory tree object that specifies the backup-identifier and encryption key of every file and subdirectory of that directory.

Problem two is solved by using the hash of the unencrypted file as the encryption key, and then storing the encrypted object in the system. Since the encrypted file is completely different from the unencrypted file, there would be no way to determine the hash of the original object from the encrypted object without breaking the encryption itself. Of course, any one else that has the same file would easily discover the encryption key by performing the hash too, but that person already has the plaintext so nothing is compromised. It seems this technique is called “convergent encryption” [8], and although it is perhaps obvious, it may be encumbered by one or more patents. I have not implemented it, but if I proceed I will need to do more research to figure out what is and what is not covered by these patents.

Implementation

My prototype was developed in Perl. My reason for this choice of language is that it is portable (it can run on Linux, Windows, and Solaris, among many others), and it is a language that fits very well with my existing skill set (i.e. I didn't need to learn a new language). However, when producing a program like this for production, portable C would be recommended, since it could then be compiled to run on Windows (the predominant desktop operating system) without the need to install an additional runtime platform.

As specified in the design, the prototype consisted of three programs. The programs can be thought of as being composed of several modules; some of these modules are shared between the programs. Here is a diagram showing names for each of the modules, and how they depend on each other.



The Main-Loop Module

An interesting aspect of the "server" program is that is completely single-threaded, which is unusual for a program that has to maintain many network connections simultaneously. All system calls involving the network are "nonblocking", so that if the remote host is not ready for the data, the server can continue processing requests from other hosts. But the server still needs some way to block for network activity. To accomplish this feat, the program uses a "Main-Loop" utility module, which I wrote. The Main-Loop module implements a loop involving a "select()" system call, which blocks until any of a vector of network sockets is ready for action, and calls a corresponding callback. This restructures the program to be completely-event based, with events like "socket-has-incoming-data" or "socket-is-ready-for-writing." Different events and different sockets can have different callbacks. When a socket has incoming data, the read() system call is invoked to get the data, and then the incoming message is processed. When a socket is ready for outgoing data, the program generates something to send and sends it.

The Main-Loop module also processes timer events, which allow the server to easily perform an action after a delay or a periodic action. For instance, the server checks whether its neighbors are still alive every two minutes.

The main purpose for implementing the server program as a single-threaded process is to simplify programming by not having to deal with thread issues, such as avoiding deadlocks or race conditions. A secondary purpose is it was a challenge for the author, who has been wanting to try something like this for some time.

The Server Program

The next module up, the “Comm” module, gets into the actual implementation of the server. It handles various communications primitives. Basically, it is responsible for serializing and deserializing packets and messages. Serialization differs between UDP and TCP, partly because they were coded at different times and partly because they are used for different purposes. The UDP packets are serialized as a verb followed by a JSON object [9]. JSON is a human readable format, similar in concept to XML. It is not tied to a particular programming language or machine architecture. This means it will be easy to implement a second program, in a different language, that can interoperate with this Perl-based prototype. TCP messages use an HTTP-style protocol. Each message consists of a method, object-identifier, and possibly some content.

Here is an example of a UDP packet, showing how it is human-readable, using JSON serialization. This is a “join” packet, which one node sends to another node when it is joining the ring adjacent to that node.

```
join {"req_id":1,"src_node":"e332fdbaeb27bf8f"}
```

The “Comm” module is also responsible for setting up the sockets when the server program starts. On startup, the Comm module creates a UDP socket and asks the operating system to assign it a port number. Then the program tries to create a TCP socket with the same port number. If successful, the program has both a UDP and a TCP socket listening on the same port number. This makes it easier for the program to communicate an address for itself to other nodes on the network. Other nodes on the network could potentially need to use UDP or TCP to communicate, so it works well to use the same port number for both. If creating the TCP socket failed, which will happen if that port number is in use, the program repeats, asking the operating system for another UDP socket with a different port number.

The “Chord” module is the next higher layer. It handles joining the ring, maintaining

pointers to other nodes on the ring, and providing an object “lookup” service. The module consists of a set of packet handlers (one handler for each “verb” it could receive in a packet) and a set of periodic tasks that get called once every time period. The packet handlers and periodic tasks are designed in such a way that it can handle other nodes going offline at any time. There are also provisions for handling the occurrence and recovery of network “partitions,” where parts of the network become inaccessible to each other for a period of time.

What happens in the case of a network partition is that the remaining nodes on each partition will consider the other nodes “missing” and construct a ring of the remaining nodes. There will be two rings, one on each partition. When full network communication is restored, the nodes will eventually discover the nodes that were on the other partition and merge the rings.

I'll jump to the Shared and Config modules next, saving the Replica module for last. The “Shared” module provides some functions that are needed from the clients and the servers. The “Config” module contains some tuning parameters and configuration settings. In this module you can control what multicast address/port the server should listen on.

The server program was designed so that the operator can specify a data directory for each instance of the program being run. In addition, the server program allows the operating system to choose the TCP port number to use. This allows any number of instances of this program to run on a single computer. This allows complete testing on a single machine, by creating any number of data directories and running one instance for each data directory. This allows testing of scalability into the 100s or more, without needing access to so many physical machines, and it allows quickly testing changes to the program, since there's only one copy of the program itself on one machine.

The Replica Module

The “Replica” module, which (if you're looking at the source code) is part of the main Server program (the server.pl file), is responsible for sending and receiving backup chunks. Since backup chunks are transferred over TCP streams, this means this module is responsible for processing TCP streams. This module processes requests from the client programs (“backup” and “restore”) and other instances of the server program (on the neighboring nodes... “next” and “prior”). Processing requests from the client programs is easy. In the case of “backup”, the request will be “POST”, which means the backup program has a backup chunk to upload to the system. The server program merely writes it to disk and returns a success code. Whether the object belongs on the current node is irrelevant, the server program will move it to the

appropriate destination afterward. In the case of “restore”, the request will be “GET”, which means the restore program knows about a backup chunk and is asking for it by name.

Processing requests from the neighboring nodes is more challenging. Requests from neighboring nodes involve replication. It turned out dealing with replication was a very challenging aspect. The design required maintaining a certain number of replicas of each backup chunk. The implementation maintains three copies. But since each workstation only has a limited view of the overall data, the workstation cannot know how many copies of a chunk exist in the network. This was solved by having each workstation track a “replica number” with each chunk. The node whose address is closest to the chunk's address assigns its copy of the object a replica number of zero. The next node then marks its chunk as replica number one, and the next marks its chunk as replica number two. The subsequent nodes, if they have a copy of that object, will then know that their copies can be deleted, and so they delete their copies of the object.

The Backup Program

The backup program runs independently of the server. It can run on the same machine as a server, or a separate machine. As a result, the initial concept of a desktop computer exchanging backup files with other computers on the network is kind of lost. The system relies on people who use the backup program to also run enough server processes that enough storage space for the backup exists.

The backup program runs a depth-first tree-walking algorithm to backup all files in a given directory. Since each individual file becomes a backup chunk, they are backed up first and uploaded to an available server node. Once the individual files are backed up, the backup program has their identifiers. For each directory in the tree, a text file is constructed listing the backup identifiers of the files in that directory, along with the name, timestamp, permissions, and other metadata for those files. (A decryption key would be included here if encryption is implemented.) This text file is then backed up in the same way as individual files. Once a directory is backed up, and its identifier is available, its parent directory can be backed up too. Eventually, all directories are backed up, including the starting directory. The backup-identifier for the starting directory is returned to the user. This identifier must be specified by the user in order to perform a restore.

The backup program, when it initializes, needs to discover the TCP ip/port of a node running the server software. It performs discovery in a manner very similar to the server program itself. The difference is the backup program only needs the address of one node. Once it has the address

of a node, it communicates with that node. As the backup program starts uploading files, it will be referred to other nodes by the node it was talking to. It may get referred to a different node every time it performs an upload.

A faster implementation could be achieved by exercising some parallelism. In a given directory, every individual file can be backed up immediately. So a faster implementation would simultaneously upload every file in the directory. This will achieve a speedup, since in a large network, each file would be getting uploaded to a unique server process.

The Restore Program

The restore program is simple in concept, and nearly as simple in implementation. It is given a backup-identifier to start from. It will fetch the chunk identified and process it. If it is an individual file it writes it to disk. If it is a directory, then it proceeds to restore every file/directory referenced in the chunk. (Again, a recursive algorithm.)

The restore program follows the same basic structure as the backup program. It uses discovery to find a node to talk to. Then it talks to that node until that node refers it to a different node. It will talk to that different node until it gets referred again. The restore program uses “lookup” requests to determine which node to restore from, and “get” requests to fetch actual files.

Speeding up the restore program is a little more challenging than speeding up the backup program. The initial fetch cannot be parallelized, since the restore program only has a single backup-identifier to work with, and it will only get additional ones once the first chunk has been downloaded. At that point, however, the restore program could fork itself to work on different subdirectory trees.

Challenges

This project was challenging in several respects. First of all, working on a distributed system is challenging. Without a central server, there is no computer with complete knowledge of the system. The programmer has a goal for how the overall system should behave. However, the algorithms coded must use the limited information and perspective of each machine to have that machine do their small part of the overall behavior. Not only is this challenging, but sometimes the programmer cannot even tell what's going on, since no system has all the knowledge.

For this project, I used two techniques to help determine how the system was behaving.

First, I designed things so that many instances of the program could run on the same computer, then actually ran it that way when doing my tests. Second, I developed some utility programs that would query each instance running on the computer, combine the information of each into a global picture, then report the result of this global picture.

The second way this project was challenging was the specific challenge of writing the replication algorithms. I mentioned above that I used replica numbers to track the replication. Well, it took several tries to get this right. What tended to happen early on was that every node would get a copy of the object, and then they would aggressively try to give each other copies of that object. Then, other times, some of the nodes would delete the object, but other nodes which shouldn't have the object, would keep the object and not do anything with it. Using the tools I developed to see the global picture, I was eventually able to sort this out.

This project was also challenging because of the unresolved problems. These are problems I was thinking about since early on in the project, and was hoping to get to start dealing with them, but some of the things I thought would be simple turned out more challenging (i.e. replication). These unresolved problems include how to backup the backup-identifier, and how to perform garbage collection.

The backup-the-backup-identifier problem is the idea that you need the full backup-identifier to perform a restore. Every time you perform a backup, you will have a different backup identifier (unless no files have changed). There needs to be a place to store the backup identifier. This is challenging. You cannot store the backup-identifier at a well-known location in the distributed storage system used by the backup system, because the storage system does not allow the uploader to specify where the file goes. Even if you could put it at a specific well-known location, the storage system would not allow you to change the contents of that file once uploaded.

My plan for the backup-the-backup-identifier problem is to add to the storage system a second namespace where the file content can be separate from the file identifier, and file content can be modified after first written. However, this makes the storage system more complicated, because now consistency and replication is harder (different systems will have differing versions of the same file). But if those difficulties can be surpassed, a backup user can create a file in this namespace that contains the backup-identifiers of all the backups that user performed. This file would be updated each time a backup is performed. When a restore needs to be performed, the file could be fetched and used to display the available restores.

The garbage-collection problem starts with asking: what happens when a server receives a file but does not have disk space for writing it. Presumably this will happen after more and more backups are performed—the system will fill to capacity. The answer: it will need to somehow free up some space by deleting old, unused backup files. But that leads to: how will the system know which files are no longer used? I've been debating for several weeks now between using a mark-and-sweep garbage collection algorithm, or a reference tracking scheme, or using some sort of time-based lease system. Only time will tell how this gets implemented.

Evaluation

My prototype has demonstrated the initial concept has a lot of potential. There are still some hurdles to get through, but so far nothing insurmountable, I think. I'll describe some of the goals I had for this project, and how well I did at achieving those goals. Then I'll discuss future potential for this project.

My primary goals included:

1. the system will be truly peer-to-peer, involving no centralized coordinator.
2. the system will allow member systems to come online and go offline at any time.
3. the system will allow a full restore, even when some nodes have gone offline or are otherwise unavailable.
4. nodes will automatically discover each other, i.e. no configuration necessary.

For secondary goals, I included:

1. nodes can specify how much disk space they want to contribute.
2. all backups will be encrypted.

I achieved all primary goals, but none of the secondary goals. The system is truly peer-to-peer. Each node is only responsible for a portion of the overall storage system, and only maintains pointers to its nearby neighbors and a few distant nodes. The system functions despite nodes coming online and going offline at any time. The only exception is that backup and restore operations can have temporary failures during the time a ring is responding to a change in number of nodes. A full restore is possible even when some nodes are not available, with a certain limitation. The limitation is that a restore will fail if the offline nodes were 1) adjacent to each other on the ring, 2) they went off simultaneously, and 3) they were the same or greater in number as the number of replicas maintained by the system. Nodes automatically discover each

other, as long as they run on the same network and the multicast traffic is not blocked by a host-based firewall.

The secondary goals were unachieved because they were not worked on. Perhaps, in the next couple of months, I will achieve those as well. From the amount of thinking I've done about them, I believe they will be fully achieved.

I've thought from the beginning that this project has a lot of potential for many organizations. Even since starting on it, I've encountered a few real-life places that could use software like this. These were small organizations (2-5 employees) that had two or more computers but their existing backup solutions were very poor or nonexistent. If this project develops into a software package that could be installed on Windows and configured to run automatically, these organizations would have a pretty decent backup solution without having to purchase special hardware. After all, they have enough disk space, through the computers they already have. If they had a system that would replicate the data automatically, they would have redundancy and therefore resilience to all sorts of failures.

Summary

This project has achieved the design of a completely distributed, peer-to-peer, desktop backup system. The backup system could provide desktop backups without requiring the purchase of a new server, arrays of disks or other dedicated hardware. The only thing required is that the desktops be on a network, and many organizations already have this.

This project's implementation was just a prototype. It lacks a reasonable user interface, and is still missing features needed for widespread usage. When fully implemented, the program could be installed on nearly any desktop system, and with very little or no effort provide full data reliability.

The project is a great demonstration of distributed system concepts. Being peer-to-peer, each participating workstation contributes just a part of the overall system. This achieves excellent scalability, but makes for a challenge in design and implementation. It is challenging to design an algorithm that does the right thing despite not having the total picture.

Bibliography

Some more information about some of the projects I've mentioned in this proposal can be found at the following locations.

1: “Norton Ghost: System Restore.” Symantec Corporation. Accessed 3 May 2008. <http://www.symantec.com/driveimage/>.

2: “Norton GoBack 4.0.” 17 Aug 2005. PC Magazine. Accessed 3 May 2008. <http://www.pcmag.com/article2/0,2817,1847391,00.asp>.

3: “BackupCoop peer to peer backup.” Accessed 3 May 2008. <http://josh.com/BackupCoop.htm>.

4: “BuddyBackup.com – Free online backup.” Databarracks. Accessed 3 May 2008. <http://www.databarracks.com/buddybackup/>.

5: Batten, Christopher, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. "pStore: A Secure Peer-to-Peer Backup System." 8 Dec 2001. MIT 6.824. Accessed 12 Feb 2008. <http://kbarr.net/static/pstore/>.

6: Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." August 2001. ACM SIGCOMM 2001. Accessed 10 Feb 2008. <http://pdos.csail.mit.edu/papers/chord:sigcomm01/>.

7: Corbet, Jon. "The guts of git." April 12, 2005. LWN.net. Accessed 10 Feb 2008. <http://lwn.net/Articles/131657/>.

8: Douceur, John R., Atul Adya, William J. Bolosky, Dan Simon, Marvin Theimer. “Reclaiming Space from Duplicate Files in a Serverless Distributed File System.” 2002. Distributed Computing Systems. Accessed 4 May 2008. <http://research.microsoft.com/~adya/pubs/ICDCS2002.pdf>.

9: “JSON.” Accessed 4 May 2008. <http://www.json.org/>.