DIBS: Distributed Backup for Local Area Networks

Eugene Hsu Jeff Mellen Pallavi Naresh

Abstract

DIBS (Distributed Intranet Backup System) is a distributed file backup system for computers in local area networks that lack dedicated or secondary data storage. DIBS exploits the unused disk space among a set of connected machines and the high speed of local area networks to provide a transparent peer-to-peer backup solution. Each node in the DIBS system contains a module that manages its list of critical files and the files it stores for other nodes, a server that responds to requests from other nodes, and a client interface to perform backup operations. Each node is responsible for its own files, and, given adequate space on the network, will ensure the satisfaction of replication invariants. The system is designed to provide complete recovery for a crashed node, recovery from transient machine failures, protection for mistaken file overwrites and deletion, and flexibility in the face of changing network-wide storage conditions.

1 Introduction

The loss of data due to hardware, software, or user error is at best inconvenient; at worst, it can cost innumerable hours and dollars of lost productivity. The common solution to data loss is backup. And yet, computer users continue losing data. The simple reason is that backup is inconvenient. For backup to be effective, data must be stored separately from the computer. This requires the use of cumbersome removable media or expensive, centralized storage with skilled administration. Many individuals and organizations can not afford the time or cost of such solutions.

To remedy this, we propose a solution that takes advantage of two trends in computing: the rapid growth of disk capacity and the increasing prevalence of local area networks. Not too long ago, several hundred megabytes of storage was considered more than adequate. Now, it is common to find consumer workstations equipped with tens or hundreds of gigabytes of data. From 1961 to 1991, average hard disk areal density has increased steadily at a rate of about 30% per year. Beginning in 1991, that rate doubled to 60% a year until 1997, when capacity began doubling annually until 2001 [3]. Density continues to improve. The average computer user, as a result, often leaves many gigabytes of unused space. At the same time, networked environments are still increasing in market penetration. Most businesses, and often many homes, have multiple computers connected by very fast local area networks.

We are interested in providing a low cost backup solution for recovery of critical data and unique files (such as media files). We are not interested in providing backup for program or operating system files which may be recovered in more cost effective ways.

In this paper, we propose DIBS, a system that utilizes unused aggregate disk space and the speed of local area networks to provide a transparent peer-to-peer backup solution for critical user files. At a very high level, our system distributes a user's files among semi-trusted peers. By querying peers, a user can reconstruct damaged or lost files. In a sense, the network of peers functions as a reliable, self-administering storage device.

We begin in Section 2 by enumerating our design goals. In Sections 3 and 4 we describe the data structures used by DIBS and the usage and behavior of the file operations DIBS provides. Section 5 uses the understanding developed in previous sections to describe how DIBS provides versioning and security. We follow with a discussion and evaluation of our implementation in Sections 6 and 7 and conclude in Sections 8 and 9 with a description of related work and possible areas of future work.

Throughout this paper we use the terms *peers*, *nodes*, *clients*, and *servers*. It is important to understand the context of these terms. DIBS functions among a set of cooperating peers or nodes. At any time, a node may act as either a client or a server. We may thus characterize a node as a client or a server, with the understanding that neither term is meant to exhaustively describe that node's functionality.

2 Design Goals

DIBS targets efficient maintenance and reliable data recovery in the face of multiple disk failures and accidental deletions. It is designed to balance the following goals.

Reliability. A backup system is useless if a user can not rely on its availability when various failures occur. Indeed, these are the moments when backup is most important. To achieve this goal, DIBS uses file replication and attempts to preserve replication invariants andy@M1

ID	Owner	IP	Space	Last Ping
M2	bill	18.228.0.10	1252M	10s
M3	carla	18.228.0.11	3329M	7s
M4	dick	18.228.0.12	501M	9122s

Figure 1: Server list for `andy@M1`, showing three other machines on the network.

andy@M1

Path	Owner	Locations	MD5
/home/andy/f1	andy	M2,M3	9ff9efd5
/home/andy/f2	andy	M3,M4	b260e1f3
/BACKUP/tmp/f3	dick	M4	ef5a13fa

Figure 2: File list for `andy@M1`. `/home/andy/f1` is a local file that is replicated on `M2` and `M3`. `/BACKUP/f3` is a replica of a file, stored for `dick@M4`.

under various modes of failure.

Transparency. A backup system should be nearly transparent under normal usage; that is, it should not interfere with the normal operation of the system. As a result, automation and efficiency are of paramount importance for operations that occur when systems are stable.

Security. Users of the system should not have to worry about others seeing or modifying their sensitive data. DIBS provides encryption and checksums to ensure privacy and integrity of data.

Simplicity. A system should solve a problem without unnecessary complexity, as this is beneficial to efficiency. DIBS does not use techniques intended for more general problems with different cost models.

These goals often conflict. DIBS attempts to achieve a compromise through assumption and design. In the following sections, we will elaborate on some of these details.

3 Component Architecture

DIBS provides functionality for various backup operations as well as security and versioning. For clarity, we postpone a discussion of security and versioning to Section 5 and describe a simplified version of DIBS. These features can be easily understood once the underlying data structures and operations have been explained.

Each DIBS node maintains a list of other currently accessible nodes, known as the *server list*. The server list, shown in Figure 1, contains a list of other nodes currently on the network. Each node builds this list by listening to broadcasts from other nodes. All nodes periodically broadcast their unique identifiers (possibly MAC addresses), their owners, their IP addresses, and the amount of space that they choose to offer for backup. In addition to storing this information, a node will store the time since the last received broadcast from other nodes. This information is used to determine whether a node has left the network.

A node keeps track of the files that it wants to back up, and the files that it is backing up for others in its *file list*, shown in Figure 2. Each entry in this list stores the local path of the file, the owner of the file, the owner's machine, and an MD5 hash of the file contents. This list functions as a cache of system state that can be rebuilt with varying levels of effort.

4 System Operation

In this section, we describe how DIBS nodes handle various operations that are associated with backup. Again, we assume no versioning and security until Section 5 for clarity. Here again we will also refer to the node performing the operation as the client and the node that responds to the operation is the server.

4.1 Primitives

We begin by describing a few primitive operations of our system. A user may call these directly, but they are also used as subroutines for more complex operations in our system.

4.1.1 Single-File Commit

The single-file commit operation either adds a file for backup or updates a file that is already in backup. In either case, it attempts to verify k replicates exist on the system, preserving k-replication.

To commit a new file to the backup system (we can check if it's a new file using the file list), the clients use the server list to identify k servers that are willing to accept the file. In our implementation, the client selects these servers randomly from the server list, with probabilities weighted by free disk space. The file is sent to each of these servers, and all file lists are updated accordingly.

If a file is already in the backup system, then the client's file list will indicate which servers are holding replicas. The client attempts to send the updated version of the file to all of those servers. If any of those servers are off of the network, then the client will randomly select new servers to preserve the k-replication. File lists are then updated accordingly.

Note that, if a server is temporarily offline, it will not realize if a client reassigns responsibility for a given file to another machine. As a result, the server will retain the file, thinking that it still has an obligation to the client to replicate it. What is important at this point is the that client knows, through its file list, where the correct replicas are stored. We handle the case of extraneous replicas in future sections.

4.1.2 Single-File Retrieval

To retrieve a file from the backup system, a client need only refer to its file list for a list of servers that hold replicas. Any one of these servers can be queried for the file, which can then be restored to the appropriate location on the client. No changes to any file lists are necessary.

4.1.3 Single-File Removal

The client may want to explicitly remove a file from the backup system, perhaps as a courtesy to others. To do so, a client sends a message to all servers that are replicating the file that indicates that it can be removed. A server, upon receiving this message, verifies permissions and performs the removal. Again, it is possible that a temporarily offline server will not receive the message and therefore hold the file unnecessarily. As in the commit operation, the client's file list will be correct, and extraneous replicas are handled separately, as described in the following section.

4.2 Synchronization

The primitive operations described in the previous section are inefficient in the case where multiple files are being committed to a server. Furthermore, it has the potential of leaving extraneous replicas that should be flushed from the system. The sync operation resolves both of these issues.

During a sync operation, the client and server exchange lists of files. More specifically, the client sends the server a list of files that it believes the server should be holding, and the server sends the client a list of files that it is holding for the client. These lists can be extracted from the file managers on each machine; however, the server should verify that it is actually holding the files it claims that it is holding.

By comparing these lists, the client and server can determine the proper actions to take. In stable state, these lists will be consistent. However, there are several important cases in which this will not be the case.

- The server may not actually have the files that the client believes that it is holding. This may occur if the user of the server decides to reclaim space by deleting some of the backup files.
- If the server is off the network, the client may decide to transfer responsbility for certain files to another machine to protect the k-replication invariant.

DIBS handles these scenarios in a unified manner. To allow for a more concrete explanation, suppose that the client and server exchange file lists $C = \{x,y\}$ and $S = \{y,z\}$, respectively. In other words, the client believes that the server is holding files x and y for it, and the server believes it is holding files y and z for the client.

The objective of the sync operation is to rectify the inconsistencies in file lists. This can be either handled by the client or the server, and is performed on a case-by-case basis.

- If file x is in C but not in S, then the client realizes that x is not properly replicated on the server. Thus, the client must update its file list accordingly. If this should cause a violation of the replication invariant, then it is the responsibility of the client to reassign the file to another server.
- If file z is in S but not in C, then the server realizes that it no longer has responsibility for holding it. Thus, the server is free to delete the file. This resolves the extraneous file situations mentioned previously.
- If file y is in both C and S, then both of the lists agree. In this case, the server simply timestamps the file to reflect the sync operation. While this does not appear necessary at this point, it is helpful information that allows DIBS to automatically expunge lost files. This will be discussed in greater detail in the following section.

4.3 Replication Maintenance

A DIBS client will attempt to preserve k-replication. If this invariant is violated, then the client will reassign responsibility to another server. In this section, we describe how this can be performed reliably.

4.3.1 Violation Detection

The replication invariant is violated if one of the k servers is permanently removed from the network. This is impossible to detect with certainty, as it is possible that its absence is only temporary (perhaps due to a reboot or an overnight shutdown). DIBS uses a simple heuristic to distinguish between these cases: if, based on the information in the server list, a server has been offline for a given period of time, then it is assumed to be permanently down. In practice, the time might be set at several days, to allow for overnight or weekend shutdowns.

There is a more subtle case in which the replication invariant is violated. Even if a server remains on the network, there is nothing preventing the user of the server from manually deleting some of the backup files, perhaps to make extra space on the disk. This case is detected by the sync operation. Thus the client must initiate the sync operation on a timely basis to maintain ensure k-replication.

4.3.2 Reassignment

When a client detects that a file's replication invariant has been violated, it will reassign that file to another server. If the file currently exists on the client machine, it can just select a new server and perform a standard single-file commit. However, if the file does not exist on the client machine (perhaps it was accidentally deleted), it is still desirable to maintain the replication invariant.

Recall that a deleted file will still have an entry in the client file list. This entry stores a list of servers that replicate the file. If one of those servers should go down or lose the file, then the client can simply fetch a temporary copy of the file and forward it to a new server, as before. This implies that it is desirable to have a replication factor of $k > 1$.

4.3.3 Cleanup

The stated reassignment scheme works if the violation detection is always accurate. However, as implied previously, it is not possible to know with certainty that a server has gone offline permanently. Consider a scenario in which a client replicates a file x on a server. The server is taken offline for a period of time, and the client reassigns responsibility of x. If the server rejoins the network, then it will still be holding x even though it no longer has any responsibility for it.

As described previously, the sync operation will handle this case. However, since the client may not be storing any files on the server, there is no guarantee that a sync will ever occur. This is why the sync operation places timestamps on each file, even if they have not been updated. If a server notices that a file has not been synced for a long period of time, then it can be deleted.

4.4 Restoration

If a machine has crashed, and its owner has lost all data, that owner can attempt to restore files and state by querying all other nodes on the network. Once the server list is updated, a client can ask all connected servers to return a list (LS) of the files it owns. After that, the node can retrieve each file to completely restore the set of backed-up files on a new or restored machine.

5 Extended Features

Equally important to the system are the issues of security and versioning. In this section we describe how these features are provided alongside the previous DIBS operations. A full assessment of the security provided by DIBS is deferred to Section 7.

5.1 Security

DIBS operates under the assumption that participating peers are semi-trusted. Communication does not take place over an anonymous network but a LAN where all participants are known to each other. We assume that if malicious participants exist, they will have more direct means to attack other users, such a physical access to other computers. Our main security consideration is thus ensuring privacy by protecting the contents and names of files backed up on other machines from being read by other users. Secondary considerations are authentication and guaranteeing file integrity.

Protecting the contents and even names of files is important to ensure the privacy of potentially sensitive documents. To this end, we use symmetric secret key encryption to encode data and filenames, where the key is a user supplied password. All filenames and file data are encrypted before being sent to a peer. Upon retrieving locally owned files, file names and contents are decrypted to reveal the plaintext name to the owner.

DIBS uses address-based authentication. It assumes that the identity of the source can be inferred based on the network address from which packets arrive. A problem with this scheme is that IP addresses may be assigned by a DHCP server, which are subject to change, depending on the lease interval of a user's IP address. Making "HELLO" messages somewhat regular would reduce the severity of this problem. The problem of IP spoofing and permissions is discussed in Section 7.

DIBS maintains integrity of retrieved files through use of the MD5 and encryption mechanism. User M_1, holding files for user M_2, may intentionally or unintentionally tamper with M_2's files. The corresponding MD5 of the file serves as a loose check against tampering. M_2 can retreive a file, decrypt it, and calculate the MD5. If the MD5 matches M_2's records, the integrity can be verified. On the otherhand, if a user looses the MD5, she will not have a point of comparison. In this case, the encryption mechanism serves as a checksum. If M_1 does not know M_2 password then M_1 cannot change M_2's file in a deterministic manner. Thus when M_2 retrieves a file from M_1 and decrypts it, if the file does not contain garbage than M_2 knows that the file had not been tampered with.

5.2 Versioning

Versioning is an important feature of DIBS. We did not implement the complex features provided in such applications as CVS but we did feel it was important to provide some level of versioning. The rationale for providing versioning is that without it, if a user backs up a file and that file is unknowingly corrupted and recommitted to backup, the user will have no way of retrieving the uncorrupted copy. The backup system will thus prove useless. In protecting against this case, we provide a simple versioning system.

The versioning system works as follows: every file manager stores a version table with a list of committed filenames mapped to a list of committed version numbers. Upon a commit, in addition the operations previously described, the file manager increments the last version number and adds the number to the list of committed versions. It then commits the file under the original filename, concatenated with a suffix indicating the version number. The version manager maintains the maximum number of versions that can be stored per file. If a commit causes the number of versions supported to be exceeded, it issues a delete on version no longer supported by the manager.

6 Implementation

Our DIBS software is divided into three primary components: the main DIBS backup module, a discovery service, and a user-facing GUI. The backup module, written in Perl, is the main component in the system, managing system logic, inter-node communication, and ensuring consistency and replication invariants across the network. The other two components act as windows into and out of the backup module. The GUI allows users to more easily manage their stored files. The discovery service provides a mechanism such that DIBS nodes can quickly and easily find each other over a local area network.

6.1 Discovery

The discovery service, `discd`, written in C++, is an agent that broadcasts basic information about DIBS nodes to the local area network. Each user, when starting DIBS, specifies a username and unique machine ID, as well as a space-available parameter to `discd`, and a rebroadcast interval. After each interval, the DIBS node will rebroadcast its statistics (as UDP packets) to the default broadcast IP address, `255.255.255.255`. At the same time, `discd` listens for similar broadcasts from other DIBS nodes, and stores every message received.

The DIBS backup module, or any other agent, can query `discd` by establishing a telnet connection over a predetermined port, specified when `discd` starts. The discovery manager will return a list containing the IP address, machine ID, owner, free space (in bytes) available, and time since the last message was received from each node. This is the mechanism by which DIBS detects crashed or disconnected nodes; if a node has not reported to the discovery service within a certain period of time, other nodes will assume it has left the network, and can choose to reassign files stored on the departed node.

6.2 GUI

Our GUI, written in Java as a Swing application, allows users to execute basic DIBS operations, such as restoring a machine from scratch, retrieving a previous version of a file, committing a file to backup, and removing a file from backup. The DIBS GUI visualizes the file list for a node, showing stored files and their locations in a table. A user can select an individual row to manipulate that record, such as committing a new version or removing that file from backup. In addition, the DIBS GUI acts as a window into the status of the network, displaying information captured from `discd` (via the backup module) in a table. A screenshot is shown in Figure 3.

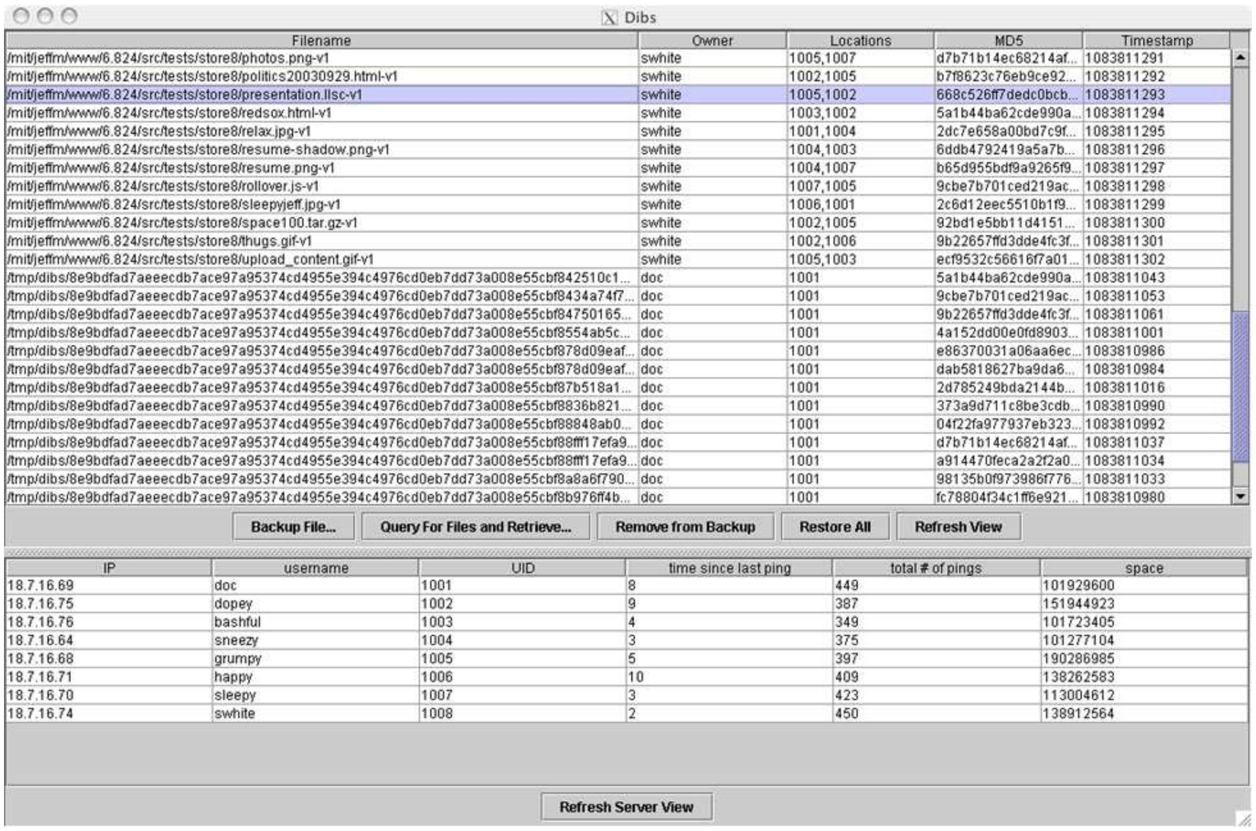
6.3 Backup Module

Ultimately, both the discovery module and the GUI rely on the backup module, which contains all logic and state for the system. It is implemented completely in object-oriented Perl, and is divided into a system manager, which maintains a list of stored files and connected servers, and a communication layer, which is responsible for sending and receiving messages to and from other DIBS nodes. The two components communicate with each other and with other external agents over HTTP.

6.3.1 System Manager

The system manager contains the file and server lists, mentioned in the system overview, an `HTTP::Daemon` instance for listening to local requests from the communication layer or GUI, as well as a version table which contains the valid version numbers for local files. It also maintains behavioral state, initially specified in a configuration file (`dibs.config`), including the maximum number of versions to maintain and the default number *k* of nodes to back up to. While this state adjusts the behavior of the DIBS module, it is the data structures that drive it.

The file table is stored as a Perl hashtable (encapsulated in `FileManager.pm`) with local filenames as keys, and 4-tuples (owner, locations of backups by machine ID, MD5, and timestamp) as value records. Because most backup operations use files as parameters, making local



Filename	Owner	Locations	MD5	Timestamp
/mit/jeffm/www/6.824/src/tests/store8/photos.png-v1	swhite	1005,1007	d7b71b14ec68214af...	1083811291
/mit/jeffm/www/6.824/src/tests/store8/politics20030929.html-v1	swhite	1002,1005	b7f8623c76eb9ce92...	1083811292
/mit/jeffm/www/6.824/src/tests/store8/presentation.llsc-v1	swhite	1005,1002	668c526ff7dedc0bcb...	1083811293
/mit/jeffm/www/6.824/src/tests/store8/redsox.html-v1	swhite	1003,1002	5a1b44ba62cde990a...	1083811294
/mit/jeffm/www/6.824/src/tests/store8/relax.jpg-v1	swhite	1001,1004	2dc7e658a00bd7c9f...	1083811295
/mit/jeffm/www/6.824/src/tests/store8/resume-shadow.png-v1	swhite	1004,1003	6ddb4792419a5a7b...	1083811296
/mit/jeffm/www/6.824/src/tests/store8/resume.png-v1	swhite	1004,1007	b65d955bdf9a9265f9...	1083811297
/mit/jeffm/www/6.824/src/tests/store8/rollover.js-v1	swhite	1007,1005	9cbe7b701ced219ac...	1083811298
/mit/jeffm/www/6.824/src/tests/store8/sleepyjeff.jpg-v1	swhite	1006,1001	2c6d12eec5510b1f9...	1083811299
/mit/jeffm/www/6.824/src/tests/store8/space100.tar.gz-v1	swhite	1002,1005	92bd1e5bb11d4151...	1083811300
/mit/jeffm/www/6.824/src/tests/store8/thugs.gif-v1	swhite	1002,1006	9b22657ffd3dde4fc3f...	1083811301
/mit/jeffm/www/6.824/src/tests/store8/upload_content.gif-v1	swhite	1005,1003	ecf9532c56616f7a01...	1083811302
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf842510c1...	doc	1001	5a1b44ba62cde990a...	1083811043
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf8434a74f7...	doc	1001	9cbe7b701ced219ac...	1083811053
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf84750165...	doc	1001	9b22657ffd3dde4fc3f...	1083811061
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf8554ab5c...	doc	1001	4a152dd00e0fd8903...	1083811001
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf878d09eaf...	doc	1001	e86370031a06aa6ec...	1083810986
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf878d09eaf...	doc	1001	dab5818627ba9da6...	1083810984
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf87b518a1...	doc	1001	2d785249bda2144b...	1083811016
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf8836b821...	doc	1001	373a9d711c8be3cdb...	1083810990
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf88848ab0...	doc	1001	04f22fa977937eb323...	1083810992
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf88fff17efa9...	doc	1001	d7b71b14ec68214af...	1083811037
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf88fff17efa9...	doc	1001	a914470feca2a2f2a0...	1083811034
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf8a8a6f790...	doc	1001	98135b0f973986f776...	1083811033
/tmp/dibs/8e9bdfad7aeeecdb7ace97a95374cd4955e394c4976cd0eb7dd73a008e55cbf8b976ff4b...	doc	1001	fc78804f34c1ff6e921...	1083810980

IP	username	UID	time since last ping	total # of pings	space
18.7.16.69	doc	1001	8	449	101929600
18.7.16.75	dopey	1002	9	387	151944923
18.7.16.76	bashful	1003	4	349	101723405
18.7.16.64	sneezy	1004	3	375	101277104
18.7.16.68	grumpy	1005	5	397	190286985
18.7.16.71	happy	1006	10	409	138262583
18.7.16.70	sleepy	1007	3	423	113004612
18.7.16.74	swhite	1008	2	450	138912564

Figure 3: The DIBS GUI.

filenames keys in the table was a natural choice. We decided to place one entry in the file table for each different stored version of the file. To prevent key collision, a unique, ascending version number is appended to each filename. In this manner, each version of a file is treated separately; it can be backed up onto different nodes, retrieved, and removed independently.

The file table contains both records for files owned by the local user, and files stored on behalf of other users. The main distinction between the two is the owner record. The owner name for files owned by the local user will be the same as the owner name stored within the system manager. The manager's logic prevents commits on files belonging to other users, as well as explicit retrievals.

We added a version table (`VersionManager.pm`) to the system manager as an optimization. The version table contains records for all files owned by a local user, mapping local filepaths to an array of valid version numbers. When committing a new version of a file, the manager checks the version table against its parameter for maximum number of versions per file. If the size of the array in the version table exceeds that maximum, it will attempt to remove the oldest version from the remote backup servers by checking where that version is stored, and issuing remote remove commands for each server.

The server list (`ServerTable.pm`) maintains the relationship between remote owners, IP addresses, and machine IDs. Like the file list, it is a Perl hashtable with machine ID as a key and the 5-tuple (IP, owner, number of pings, time since last ping, available storage (in bytes)) as a value. The manager consults this list in order to determine which server it can and should deliver a message to. For example, when retrieving a file from a remote server, the manager must look up the locations field from the file list to get the ID of the machines on which the file is stored, and then retrieve the IP address for that server to determine where to send the `retrieve` HTTP request.

In addition, the manager uses the space fields in the server list to determine where a file should be stored. If a new file is committed into the system, the manager will check the server list to determine which servers have sufficient space. It will then choose among those servers in a random manner, weighted by the relative amount of disk space a remote server has. In this manner, each DIBS node should receive approximately the same amount of load relative to how much space they offer.

The version table and file list are recoverable using the `restore` operation, but as long as a node has not crashed, it is stored locally on stable storage, using the Perl `Storable` module. In this manner, a user can shut off his/her machine and have a stable record of which files he/she has backed up on the system, without having to query a potentially incomplete network.

6.3.2 Communication & HTTP

The communication layer acts as an intermediary with other nodes. It services requests from other nodes, makes requests to remote nodes, and forwards commands onto the system manager. HTTP acts as the glue between the communication layer and the system manager, and between the two components and any external agents, such as the GUI. Both the system manager and particularly the communication layer are designed to be multiprocess (although time constraints prevented us from making the system manager MP), in order to concurrently serve backup requests from multiple machines. Thus, we needed some form of interprocess communication, as the manager and a server process may be in different contexts. HTTP, although not as compact as System V, gives us the side effect of easily supporting GUI or commandline modules to steer DIBS' operation, while facilitating this communication.

6.4 Example: Commit

To show how all the parts fit together, consider a backup (commit) instigated by the Java GUI. The GUI forwards an HTTP POST message to `http://localhost:16824/backup_commit`, with the local filename to store in the header, under the key `Filename`. The local system manager $M1$, listening on port 16824 (in a `select` loop), receives this message, and calls its message handler. This message handler disassembles the request, checks the size and availability of the file, and then picks k servers to back the file up to.

Files are backed up in two steps; first, a candidate server is selected based on the weighted-randomized algorithm mentioned earlier. The manager $M1$ issues a request to the `/canPut` resource on that server $S2$. The remote server manager $M2$ checks its file list to see if there is a naming collision or insufficient disk space. If $M2$ and $S2$ return a `OK` (HTTP 200) response, the local manager $M1$ uploads the file to $S2$ using an HTTP POST to the `/put` resource, with the (potentially encrypted) file as request form-data content. Once the file is backed up to k servers, or all candidates have been exhausted, the manager updates the file list with a new entry. The key of this entry is the local path of the stored file plus a version number. The file list also will contains the MD5 of the saved file, confirmed successful backup locations, and the current time. It also updates the version table as necessary. If all is successful, the local system manager will finally respond to the GUI with an `HTTP OK` response, and the GUI can acknowledge the backup. If at any point the operation encounters a fundamental error, an `HTTP 500` error will be generated, either by the local or remote system manager, and the error will be propagated to the GUI. This event-driven pattern is prevalent in most DIBS operations, although the responses can be more complicated.

Number of servers up	Number of files accessible w/o reassignment	w/reassignment
7/7	53/53	53/53
6/7	53/53	53/53
5/7	52/53	53/53
4/7	47/53	53/53
3/7	38/53	53/53
2/7	27/53	53/53
1/7	20/53	53/53

Figure 4: Maintaining reliability through reassignment, with light load and ample storage space on network.

7 Testing & Evaluation

We performed several tests on DIBS to test its reliability and functionality given changing network conditions. Our default environment was MIT's cluster of dialup shells (eight in all) as the default LAN, and a 15.9MB working set of files as a sample backup set. We verified the correctness of our implementation and performance of the system using these servers and the sample backup load. In each section below, we will state a design goal and show how our implementation met (or tried to meet) each one.

7.1 Reliability

We tested restoration, backup and retrieval of individual files and our entire working set, using multiple versions and storing on multiple servers. Our implementation meets the semantics required of each operation. By default, the latest known version of a file in the network is restored to a crashed node, although the complete version table and file lists are completely reconstructed from the information maintained by other DIBS nodes.

Reassigning files in the face of a node outage was the key to ensuring the availability of files and reliability of the system. We chose a value k of 2 for the default number of servers to back up to. Given enough storage space across the network (our total available storage space for the test was 1GB), this makes all files fully recoverable in the case of a single node failure regardless of reassignment. However, with multiple node failures, backups became less and less available, as shown in Figure 4.

We decided to turn on file reassignment after 30 seconds of a node outage to simulate network balancing over time. The DIBS communication layer times out from a `select` after 30 seconds, and checks the latest server topology from the discovery service, `discd`. If it finds that a remote node has not reported in the last 30 seconds, it orders a reassignment. The system manager will then commit the file to another server. When nodes were allowed to reassign, the availability of all backups was 100%.

There is a potential problem with our implementation for reassignment, and general version management. Old versions are stored on other machines for retrieval and error correction, but they are not stored locally. Thus, if a reassignment occurs, the system manager will commit the latest stored version of a file, and not the last most-recently-committed version. If this happens, the version number of the file will be updated to reflect proper version histories, but a volatile network (and low reassignment time) can cause the inadvertent replacement of explicitly committed versions. We view this as a tradeoff to having at least one recent valid version of a file reside somewhere else in the network.

7.2 Space Conservation

One of our goals was to make sure that not too many old replicas and versions of backups existed within the DIBS network, as space is a limited resource in such an application. Our main weapon against excess replicas is the same syncing `confirm` primitive. If node N_1 presents node N_2 with a list of files N_1 thinks N_2 has, and $N2$ has a file not in N_1's list, N_2 should delete that record (and backup file). We noticed that this case may arise if N_1 had crashed and was recently restored, and N_2 was absent for the restore request. Thus, we decided to add a "seen list" to the system manager. If N_1 has not yet seen or talked to (restored from) N_2, then it will not sync with N_2. Thus, there will be no sync to trigger the deletion of files on N_2 that really should be on N_1.

7.3 Performance

We incur moderate overhead over direct file transfer by computing MD5s in each file operation, operating over HTTP, transferring unchunked messages for file uploads, and likely because using Perl for transport and processing is slower than system calls. (Our primary motivation for choosing Perl was access to existing modules.) Transferring the 15.9MB dataset between two AFS servers took 23 seconds using `scp`. Backing up the same dataset to two machines took 93 seconds, and retrieval took 58. Perhaps making the manager multi-process and enabling chunked HTTP upload will increase speed. However, adopting a conservative reassignment and storage policy may be the best ways to reduce network chatter.

7.4 Security

In our implementation, we provide filename encryption, but have not gone as far to encrypt the contents of the files. This was postponed for ease of testing and is a small fix. A production version of DIBS would be sure to include this feature.

Assuming file encryption, data backed up by DIBS is less secure than email systems or most password protected systems because these systems are often protected from dictionary attacks. Encrypting data with a password, as DIBS does, is vulnerable to brute-force dictionary attacks. Our alternative was to use private and public key encryption, but the private key in such a key pair may be lost in the case of a machine failure. This key would need to be stored separately on a removable media device, which unless kept in a secure place would be susceptible to theft. A lost private key would render all file data (and filename data) encrypted with that key unreadable, and theft of a private key may be a more likely attack than a dictionary attack. In order for DIBS to be adopted by users, we feel it is important that DIBS be simple and easy to use. We believe private/public key encryption significantly increases the burden upon users. Thus for the above reasons, we choose secret key encryption.

DIBS is no more susceptible to eavesdropping attacks as it is to dictionary attacks. An eavesdropper can obtain a user's encrypted files, but so may any other legitimate user of the system. Without the password, an eavesdropper cannot decrypt the files. DIBS is susceptible to known plaintext or chosen ciphertext attacks. If user M_1 has both the plaintext version of and ciphertext version of M_2's file, he may be able to deduce M_2's password.

DIBS uses address-based authentication. The process of storing machine ID/username/IP pairs in a server list is however susceptible to IP spoofing. We are not very concerned with privacy in this case as this is no different thatn an eavesdropping attack. However if a user launches an IP spoofing attack, she may trick users into believing their files are safely backed up. An alternative to address-based authentication is a cryptographic authentication protocol, such as certificate based authentication. These protocols are often much more secure than address-based authentication, but are significantly more complicated, requiring the use of such things as certificate authorities or public and private key pairs.

Message authentication is also a problem. A user may announce themselves as a person they are not and thus issue commands on behalf of another user. We recognize this as a significant security concern. A solution to this problem is to require all messages be timestamped (to avoid replay attacks) and signed by user specific private key in order to be verified by the corresponding public key. Again, this requires issuing and storing the private key on a secure, removable device.

8 Related Work

Our system is similar to pStore [1] and Pastiche [2], which are both peer-to-peer backup systems. They both have similar design goals as well. Pastiche builds its system atop the Pastry infrastructure for object location. pStore uses Chord, a distributed hash table implementation, to reliably store data. Instead of using such external overlay networks and location services, which are intended to abstract the location of a particular node from a high-level system [2], we assume the use of an intranet or local area network. Such networks are more static and allow for broadcast discovery and tracking of nodes. With this assumption, we can make optimizations for greater efficiency and simplicity in our system.

pStore does not actively try to protect replication invariants. As with our system, file data is kept on multiple machines. pStore relies on probabilities to ensure that at least one peer holds file data. Thus, it is unlikely that all files can be retrieved if a small number of peers fail. Our invariants are similar to those of Pastiche, if a peer holding a client's backup files should fail, it becomes the responsibility of that client to ensure that the file data is redistributed to another machine (during the creation of a new snapshot). After much discussion, we elected to use this strategy in DIBS. If a peer node goes down, the client must find a new node to backup its files to.

An alternate strategy we discussed would be to maintain an invariant that two copies of a file must exist in the network, if space allowed. Thus, nodes would be responsible for the files of other nodes. If a client crashed, a node would detect that it was the only one with a copy of the bytes, and copy accordingly. This defends against one type of failure ours does not: sequential catastrophic crashes. If node N1 loses its data, all nodes with N1's data would copy to another node. If a second node (N2) crashes, the data still exists, up to the point where it is no longer possible to keep copies of all backed up files on the network. In DIBS, the data backed up on N2 by N1 may be lost, unless a user executes a restore operation on a different machine. However, we decided that the potential for widespread copying in the face of transient outages outweighed the benefits of protecting against that particular outage, especially since in the face of complete machine failure, a user will likely want to restore on another machine as soon as possible.

It is also important to discuss things that we do not do. pStore utilizes a sophisticated versioning system to avoid redundancies (in a sense, an incremental backup). We note that, for many types of files, this may not be effective. Compressed files such as images, for instance, can change structure completely with small changes. We leave the task of complex versioning to specific applications. CVS is an established system for any sort of text. Microsoft Word stores in each document a log of changes. Instead, each DIBS node supports simple versioning to protect against accidental overwrites and deletions.

9 Conclusion

The strength of the DIBS system is that it is a relatively lightweight, transparent solution to the problem of backup. It requires little maintanence from the user and unnecessary network chatter.

Further work should be done within the realm of security. DIBS currently provides no strong form of authentication. Fairly standard authentication techniques can be applied to DIBS in a straightforward manner. However, this increases the complexity of the system and burden upon the users. Thus, it is important to understand the needs of the potential users and what security measures are necessary for users to adopt the system.

DIBS does not address timing and scheduling issues, such as when synchronization actions should occur and how long a server must be down before it is assumed to be dead. These parameters are again user dependent and should be set after studying user needs and habits. Just as replication and maximum version parameters are stored in the system manager, default timeout times can be added as well.

Finally, DIBS can go faster. By chunking uploads and converting the system manager into a multi-process component, DIBS will achieve better performance in both one-to-one and one-to-many communication. However, making the manager a multi-process component will require more advanced synchronization methods, and we would need to be careful not to introduce inconsistencies in the file list.

References

[1] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system, December 2001.

[2] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *In Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*, dec 2002.

[3] Steven J. Vaughan-Nichols. Hard drive technology reaches a turning point. *IEEE Computer*, 36(12):21–23, 2003.