Designing a Self-Maintaining Storage System

Satoshi Asami, Nisha Talagala and David A. Patterson*
{asami,nisha,pattrsn}@cs.berkeley.edu
Tertiary Disk Project
Computer Science Division
University of California
Berkeley, California
http://now.cs.berkeley.edu/Td/

Abstract

This paper shows the suitability of a "self-maintaining" approach to Tertiary Disk, a large-scale disk array system built from commodity components. Instead of incurring the cost of custom hardware, we attempt to solve various problems by design and software. We have built a cluster of storage nodes connected by switched Ethernet. Each storage node is a PC hosting a few dozen SCSI disks, running the FreeBSD operating system. The system is used as a web-based image server for the Zoom Project in cooperation with the Fine Arts Museums of San Francisco (http://www.thinker.org/). We are designing a self-maintenance extension to the OS to run on this cluster to mitigate the system administrator's burden.

There are several components required for building a self-maintaining system. One is decoupling the time of failure from the time of hardware replacement. This implies the system must have some amount of redundancy, and has no single point of failure. Our system is fully redundant, and everything is constructed to avoid a single point of failure. Another is correctly identifying failures and their dependencies. The paper also outlines several approaches to lower the human cost of system administration of such a system and making the system as autonomous as possible.

(The table of contents are provided here for your convenience only; it will not appear in the final version of the paper.)

*This research was funded by DARPA Roboline grant N00600-93-K-2481 and the State of California MICRO program. We also thank IBM and Intel for donating disk drives and PCs, respectively.

1 Introduction

Maintenance is a big problem for large disk-based storage systems. For instance, a 1993 survey by Strategic Research Corp. shows that the annual cost of system administration is almost 3 times that of the annual cost of hardware[1]. Also, John Wilkes has mentioned recently that there is a 2–12 factor of difference between storage and management, depending on the definition of management[2]. In addition to the cost, proper maintenance is critical for storage systems because the consequence of a failure can be loss of valuable data. Packaged solutions such as hardware RAID boxes and turnkey server systems such as Tandem[3] and Network Appliance[4] have been successful in the marketplace because of these reasons. However, this approach does not scale well past the terabyte mark, both in price and maintenance complexity, as having to purchase multiple of these boxes greatly increase the complexity of administration.

Our system uses commodity hardware to solve these problems[5]. We have built a cluster of storage nodes to be used as a web-based server. The storage nodes are connected by switched 100Mbps Ethernet internally and have an ATM connection to the outside world. Each storage node is a PC hosting a few dozen SCSI disks, running the FreeBSD operating system. The system is used as a web-based image server for the Zoom Project in cooperation with the Fine Arts Museums of San Francisco[6]. We are designing a self-maintenance extension to the OS to run on this cluster to mitigate the system administrator's burden.

There are two aspects of the system we have to consider when discussing administrative issues: the system administrator's perspective, and the user's perspective. They will be discussed separately.

For the system administrator, it is clearly desirable that the system continue functioning in the administrator's absence. However, all systems require human intervention at some point, for instance, for hardware replacements, so there is no system that is fully automated in the sense that there is no human system administrator present at any time. By "self-maintaining", we mean that our storage system will simplify the system administrator's job in the following ways:

- Maintenance will only be required at fixed intervals.
- At maintenance time, the required tasks will be clearly defined.

In addition to reducing the cost, this type of maintenance will also reduce the chances of operator error by allowing the person to work under less stressful conditions. Not having to repair the system right away, as the system is fully functional even without the repair, reduces the mental stress. Also, an operator working during regular working hours is less likely to make mistakes than one working at 4AM.

It is essential to give the impression to the end users that the system is 'up' at all times. However, our users are connected to the system over the Internet, and thus are used to hitting 'reload'. If we can re-route all accesses within a few seconds of a failure, the user will not be able to distinguish a transient network problem on the Internet with a failure on our system. In fact, given the unreliability of the Internet, most problems will be caused by the network.

There are several components required for building a self-maintaining system. One is decoupling the time of failure from the time of hardware replacement. This implies the system must have some amount of redundancy, and has no single point of failure. Our system is fully redundant, and everything is constructed to avoid a single point of failure. Another is correctly identifying failures and their dependencies. We are planning to add several features, such as enclosure monitoring, to Eric Anderson's CARD system[7], an extensible monitoring system based on relational databases, and use it on our servers.

The last major component is repairing. There are three steps necessary for this; to mask failures so the system can continue functioning, to take actions to remove the vulnerability caused by a failure (e.g., reconstruct contents of a failed disk) and to prepare precise repair instructions for the human operator.

The remainder of the paper is organized as follows. Section 2 described the related work in the field. In Section 3, we clarify the concept of self-maintainability and explain how we approach the problem. Section 4 describes the application briefly. In Section 5, we illustrate the architecture of the system we are using for the experiment. Section 6 illustrates the software architecture of our system, and Section 7 explains our validation methodology, and Section 8 describes how the system can be scaled to a much bigger size.

2 Related Work

There are related work in high-availability servers, network attached storage, and maintenance of large clusters.

2.1 High-availability servers

Tandem manufactures network servers[8, 9]. Their NonStop servers are fully redundant and have hot-swappability of most components, thus it is not necessary to take them down even during repairs. Their architecture, called "shared nothing", has no single points of failure. Any one component can fail and the system will still function. Their system administration suite is called Tandem Maintenance and Diagnostic System (TMDS)[10]. It has an auto-diagnostics feature that uses an AI program that compares the symptoms of the system to known failure modes and tries to identify the failures. It can optionally dial up Tandem technical support with a report which a remote technician can consult while running diagnostics of his own.

Network Appliance[4] sells network servers built around Digital's Alpha chip. Their NFS servers are advertised to outperform a 4-way P6-200MHz Windows NT system by 2 to 10 times. They use RAID 4 (block-interleaved parity with a dedicated parity disk) with NVRAMs to increase performance. They have a filesystem that can recover from crashes very quickly.

2.2 Network attached storage

Microsoft Tiger is a video server built from commodity PCs (called "cubs")[11, 12]. Their goal is to tolerate the failure of any one cub or disk without degradation of service. They use mirroring for backing up data. This is because they cannot tolerate the runtime reconstruction overhead of parity. They locate the "primary" copies of the data in the outer tracks of the disks for better performance, and the backup copies are declustered to avoid hotspots during failures. Tiger distributes all files across all disks on all cubs for maximum striping performance, but this method has a drawback of having to reconstruct data on the entire system when a new disk is added. This

reconstruction is not very expensive as there is no parity calculation involved.

A single controller serves as entry point from clients. No image data passes through the controller so it is not likely that they will become performance bottlenecks. They take great care to ensure that sufficient bandwidth is available during the entire course of the playback, and will delay start of playback if necessary. They use a distributed schedule management protocol for scalability. Each cub has a partial, potentially outdated view of the schedule and they pass the schedule around, updating it along the way. Cubs use deadman protocol for fault detection—each cub sends periodic ping to the cub on its right.

Petal[13] is another example of a network attached storage system. It is a collection of distributed servers, each containing multiple disks. They use a method called *chained declustering*[14] to avoid having the load increase 100% on a machine when its mirrored counterpart fails. Petal is a block server. The authors of Petal have also designed a distributed file system called Frangipani[15] to run on top of Petal. CMU's NASD (Network Attached Secure Disks)[16] is another example of network attached storage. xFS[17] is a combination of network striping and LFS (Log-structured Filesystem)[18].

Our system avoids the complexity of distributed filesystems by using HTTP redirects and IP masquerading to distribute user requests and mask failures.

2.3 Maintenance

Eric Anderson of the NOW (Network of Workstations[19]) Project has been studying several aspects of system administration of clusters. His most recent work introduces a system called CARD (Cluster Administration using Relational Databases)[7]. He proposes using relational databases to build an extensible monitoring system. It uses a hybrid push-pull protocol to collect data from individual machines. His emphasis is on monitoring and diagnosis. We are planning to use CARD as a building block for our monitoring and diagnosis system.

The Storage Systems Program at HP are working on a system called "self-management" of storage[20]. Their focus is on automatic assignment of storage devices; humans do not have to worry where to put what. Their prototype system is capable of assigning several thousands of objects to devices in a few minutes.

Sun's Jini[21] makes devices, including disks, identify themselves as part of their automatic component discovery paradigm.

3 Self-maintaining system

In this section, we will define what "self-maintaining" means in the context of large disk-based storage systems, and outline the requirements on how to construct such a system.

3.1 Definition

There are two aspects to self-maintainability of a system, depending on whether perspective is that of the system administrator or that of the user.

3.1.1 System administrator's perspective. To the system administrator, a self-maintaining system is one that does not require constant attention. For the purpose of our research, we define it as a system that has the following characteristics.

- Maintenance will only be required at fixed intervals.
- At maintenance time, the required tasks will be clearly defined.

What comprises a reasonable interval is an interesting issue in itself. We are planning to run some simulations after gathering enough data on failures to see how well the system will stay up with varying intervals. For the purpose of this paper, however, we assume an interval to be one week or so.

There are two major benefits of this approach:

Reduce operator errors It should help the operator's performance. Aside from not required to make repairs at odd hours, not being under the pressure of having to fix the system *right away* will very likely reduce the chances of operator errors.

Reduce operator cost Another benefit is that it is not necessary to pay a full-time wage for a system administrator of such a system. We can either hire someone part-time or have the same person administer many systems instead of just one or two.

The second reason is similar to the motivation behind Tandem's TMDS[8, 10]. Tandem has engineers at the support center 24 hours a day, and when systems are having problems, they automatically dial up Tandem to contact one of the support persons. A few hours after the failure, the support person will show up at the customer's site with necessary repair parts. Thus, the companies that purchase support contracts do not have to hire operators by

themselves; they are in effect "sharing" the operators with other companies through Tandem.

Our system will take this one step further. There is no need for anyone to be at anywhere in the middle of the night; in fact, there is no need for anyone to be at any central support center. The operators can travel from one customer site to another throughout the week, or they can be doing something else during most of the week.

3.1.2 User's perspective.

There are two classes of users on a web-based storage system like ours.

End users These are the people who use the system from the Internet. They know nothing about the internals of the system, will easily get annoyed if something doesn't work, and may not come back at all if they are unhappy. They usually only issue reads to the system.

Content providers There are relatively few people whose job is to update the contents of the web server. They are part of the project, and can be asked to wait for awhile if the system cannot allow writes at the moment. They have a good knowledge on how to use the system, but usually do not know anything about the internal workings.

To reduce end-user frustration, it is important to reduce the system's down-time as seen from across the Internet. Note that a system that relies on an operator to keep it running is not as available as one that maintains itself, as it will take minutes or maybe even hours for the operator to actually be able to repair the damage[22]. Our goal is to have the system repair any interruption of service within a few seconds, and continue to function unattended until the next scheduled visit by the operator. For a web server application such as the one we are running, this is illustrated by the slogan "repair by reload".

As Mary Baker said in her Ph.D. thesis, "in the limit, as recovery time approaches zero, a system with fast crash recovery is indistinguishable from a system that never crashes at all"[23]. Her research was about file servers on distributed operating system, and the above statement was qualified that it is only appropriate in systems that can tolerate short periods of down-time, as a cluster of workstations in a typical engineering or research environment.

I believe our application, a web server, is another example where a system with fast crash recovery is just as good as a system that never crashes. This is based on the observation that since the Internet is so unreliable, it will be impossible for the user to distinguish problems on our servers from the daily transient problems of the network. Our goal is to recover from any single failure within 5 seconds. We believe that is less than the time an average user notices a network problem.

For the content providers, the situation is a little different. It is permissible to have the system be in a state that it cannot receive input from them for a short while. However, repeated problems will affect their productivity as well as delay the update of the contents, so it is desirable to keep downtimes as short and as infrequent as possible.

In order to partially shield problems from the content providers, we offer a *portal* to which they can upload the new images. Once the data is in the portal, they can go on to their work. However, if there is a problem with some other part of the system that disallows writes, the new images may not be available on the web until the problem is fixed.

3.2 Requirements

There are several requirements for building a self-maintaining system. Here the requirements will only be listed; the implementation details can be found in Sections 5 and 6.

No Single Point of Failure Such a system is not allowed to have any single point of failures. This will make it possible to decouple the time of repair from the time of failure, allowing the system to run under an existence of a failure. Clearly, depending on how many failures the system should tolerate, it may be necessary to have more redundancy, but not having a single point of failure is a minimum requirement for any system that is designed to function continuously in an event of a component failure.

Constant and Reliable Monitoring The system should be constantly and reliably monitored so corrective actions are taken very quickly after failures. We use very simple shell scripts for monitoring, and have an interval of 5 seconds of sleeps between monitoring. The monitoring scripts are autonomous, so a failure on one machine will not cause a monitoring script on another machine to malfunction.

There are some characteristics in our application that makes it easy to build a self-maintaining system. Although these are not hard requirements, they nonetheless have helped us simplify the design of the system.

End-users issuing only reads One important aspect of our system is the read-mostly nature of end-user accesses. We believe this is not unique to our applica-

tion; many other applications with similar terabyte-capacity scale share the same characteristics. By not having to allow writes in degraded mode, this has made the immediate recovery only a matter of locating the backup and rerouting user requests there while more lengthy recovery procedures can take place.

Little internal communication The application needs very little internal communication to handle user requests (see Section 6.1 for details on how user requests are handled). This simplifies the recovery as there are only few messages or connections that might be lost due to a failure. Also, this will enable the system to scale up nicely in the future.

4 The Application

The main application for the Tertiary Disk prototype is an image database holding pictures of objects of art. The "Thinker" site (`http://www.thinker.org/`), run by the Fine Arts Museums of San Francisco, has been providing access to art images through the Internet since October 1996. They implemented a searchable index, through which the user can query their database for keywords (artist, title, description). The user will then be presented with a page of thumbnails of images that fit the search criteria, and by clicking on the thumbnails they can view larger versions of the images. The largest images they provide are about 500 pixels on one side. The original files were much larger (up to 3,096 × 2,048 pixels) but due to disk space constraints and lack of a way to adequately present them to users over the Internet, they decided to only provide relatively small images.

4.1 GridPix

Understandably, one of the most common complaints from their users was "can't you make the pictures bigger?" We provide disk space for larger versions of the images. In addition to providing the space, we also wrote a viewer called GRIDPIX[24] which is a tiled, layered JPEG format with multiple resolution levels. There are about 70,000 images occupying about 2.5TB of storage space when fully mirrored. Each images are stored in several different formats, from the original PhotoCDs, which are about 5MB per image, to human-processed TIFFs, which average 12MB per image, to GRIDPIX, which are about 1.2MB per image.

Figure 1 shows a three-layer GRIDPIX file. Tiles 1 and 2 form the smallest layer; tiles 3 through 8 are the middle layer and the largest layer consist of tiles 9 through 23.

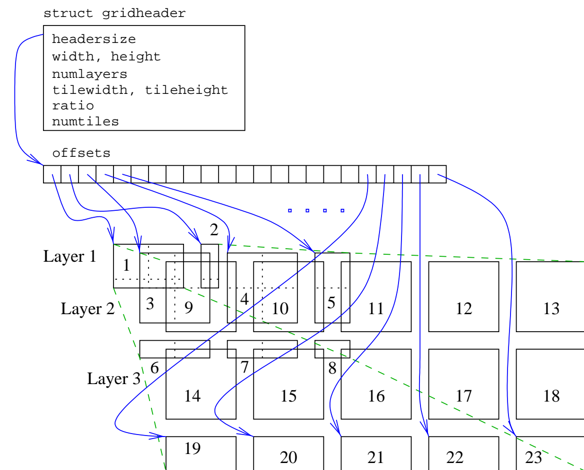

Figure 1: Three-layer GRIDPIX Image

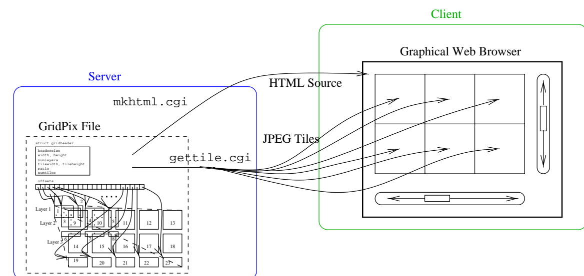

Figure 2: GRIDPIX Interface

Figure 2 shows the interaction between the client and the server. When the client requests an image, a CGI script (**`mkhtml.cgi`**) returns an HTML page describing the page layout. The individual image tiles are retrieved by a separate CGI script (**`gettile.cgi`**). GRIDPIX is completely HTML-based, so any graphical web browser can function as a client.

One thing to note is that the GRIDPIX server requires very little processing power during runtime. The images are already divided up into tiles by the TIFF-to-GRIDPIX converter, so the numerous calls to **`gettile.cgi`** only require two accesses to the file; one to get the offset and size of the tile from the header, and another to retrieve the tile itself. In particular, there are no JPEG encoding/decoding required on the server side. Moreover, the GRIDPIX files are very compact, so after a few accesses, most of the requests will be handled by on-memory disk

cache, not the disk surface.

4.2 Status

The site has been open to the public since March 2, 1998 with about 20,000 images[25]. We currently have over 70,000 images available. The reason why we couldn't simply make all images available is because the photographs are not of good enough quality to be presentable, and require manual work to crop, reorient and color correct them before they can be provided to visitors. According to the people in the museum, the response from the public has been very favorable. Also, we have not experienced any down-time yet despite individual machines crashing or being rebooted many times.

5 Tertiary Disk Architecture

The Tertiary Disk group has built a prototype disk storage system. Now we are building a self-maintaining system on top of it. Its total capacity is 3.2 terabytes. Here are some highlights:

- 20 PCs (200MHz Pentium Pro with 96MB of memory each) as disk servers
- 396 8.4 gigabyte IBM DCHS-09Y 7,200RPM Ultra-Wide SCSI disks
- 44 Adaptec 3940UW twin-channel Ultra-Wide SCSI adapters
- 52 Trimm Technologies model 381 8-disk SCA enclosures with serial interface
- 2 16-port 100Mbps fast Ethernet switches
- 4 24-port serial terminal servers for PC consoles and disk enclosure interfaces
- PCs run FreeBSD operating system, with minor modifications
- Double-ending (Figure 3) with feedthrough terminators for high availability
- Redundant frontends use HTTP redirect to route user requests
- 6 uninterruptible power supply (UPS) units
- Remotely-controllable power switches on all PC and enclosure power cables
- 2 PCs (133MHz Pentium) as HTTP frontends
- 2 PCs (200MHz Pentium Pro) as infrastructure servers (NIS etc.)

It occupies 7 racks, each 7 foot tall and 19 inches wide. There are two different configurations. 4 PCs are configured in a "disk-heavy" configuration, with 70 disks per 2 PCs. The remaining 16 are in a "CPU-heavy" configuration, with 32 disks per 2 PCs.

Only one of the computers is connected to a video monitor and keyboard; the serial ports act as consoles for the rest. One of the frontend machines has a standard monitor and keyboard to provide access to the system when we are in the machine room.

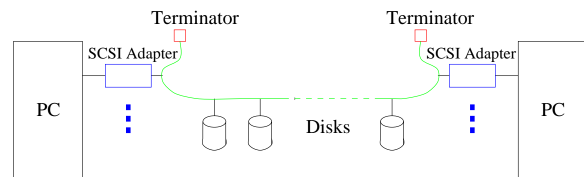

Figure 3: Double-Ending of SCSI Disks

Here are the characteristics of the system.

5.1 Commodity Components

The most important feature is that this whole system is built only from commodity, off-the-shelf, components. This lowers cost of storage by factors of two to four compared to standard RAID boxes. At the time of construction of the system (summer of 1997), large RAID arrays cost about 60 cents per megabyte; our system cost about 20 cents per megabyte using street prices of components.

5.2 Redundancy

In designing the TD prototype, we have taken care to ensure it does not have any single point of failure. Below is the list of places where we avoided creating single points of failures.

- Multiple UPS units provide power to the racks. There are power rails on either side of a rack, connected to different UPS units. UPS units also provide 10 minutes of standby power to survive temporary glitches in power.
- Double-ended PC pairs are connected to different power rails. They are also connected to different

network and serial switches. We use external feed-through terminators so the SCSI bus integrity is preserved even when one of the PCs completely loses power.

- Each enclosure have two power supplies. They are connected to power rails on opposite sides of the racks. Each power supply has a built-in fan, and there is a third fan in the enclosure so the airflow around the disks will not be reduced to half when one power supply fails.

- Most data is mirrored. Stable data, not necessary for the system's day-to-day operation, is backed up on tapes or CD-ROMs.

6 Software Architecture

There are several aspects of software architecture that needs to be discussed.

6.1 Handling user requests

Our main application, as described in section 4, is a web server for fine art images. The request for an image from a user first comes in to a frontend machine. The frontend will look up a table and returns an HTTP redirect message to the user's client, which subsequently connects to the backend machine holding the image. From that point on, the client interacts with the backend machine directly until the user has finished browsing the image. Figure 4 illustrates how requests are passed around.

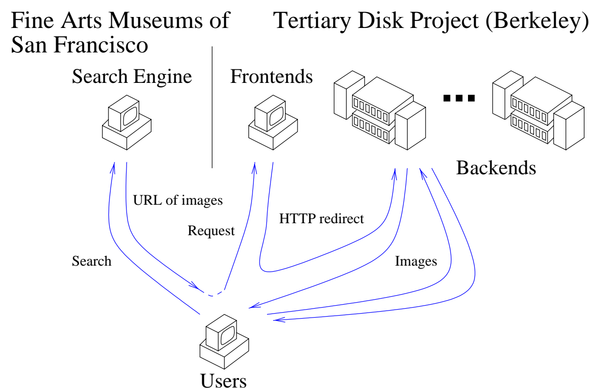

Figure 4: Handling user requests

6.2 Masking failures

There are several levels of masking that can be done to handle failures. In one extreme, a system can have non-volatile RAMs holding state information for open TCP connections so a machine crashing and rebooting will not cause any connection to be lost. Another, like NFS, is to have a stateless server with clients retrying until the server replies. This model causes the client to lock up until the server comes back up, but will not lose any requests.

Those two kind of models are required in a local-area network environment where correctness of response is valued over anything. Our premise, as mentioned in section 3, is that since the Internet is so unreliable, it doesn't make sense to try to mask 100% of the failures. Our goal then becomes how to implement the "repair by reload"; in other words, how to make sure the system will be able to mask any failure within a few seconds to allow reads from end-users.

Figure 5 illustrates how we mask machine or SCSI failures from users. We have two frontends, `tarkin.cs.berkeley.edu` and `ackbar.cs.berkeley.edu`, backing up each other using IP aliasing. The canonical address, `gpx.cs.berkeley.edu`, is usually an alias of `tarkin`. The other machine, `ackbar`, checks over the Ethernet to see if `tarkin` is up every 5 seconds. When it can't find `tarkin`, it will take over `gpx`. It will still keep checking for `tarkin` every five seconds, and will release the `gpx` alias as soon as it finds `tarkin`.

Both `tarkin` and `ackbar` check all the GRIDPIX servers every 5 seconds. This is done by fetching a file from the machine; the file will not be available if the HTTP server is down or the disk is having problems. When it discovers problems on one of the machines, it will automatically forward any subsequent requests from users to the backups. Since the CPU load on these machines, as well as all the servers, are very low, and the scripts are very simple, it is very unlikely that these scripts do not detect problems in a prompt manner.

The above two tricks will cover most cases except for one—users already in a GRIDPIX session when a server goes down. There are two ways to mask this case: to do something similar as the frontends, having two servers back up each other, or have another machine possibly one of the frontends to take over the IP address temporarily and forward the request to the backup. We are planning to implement the latter.

6.3 Operating System Support

In this section, we will describe what part of the operating system we had to modify in order to build our system.

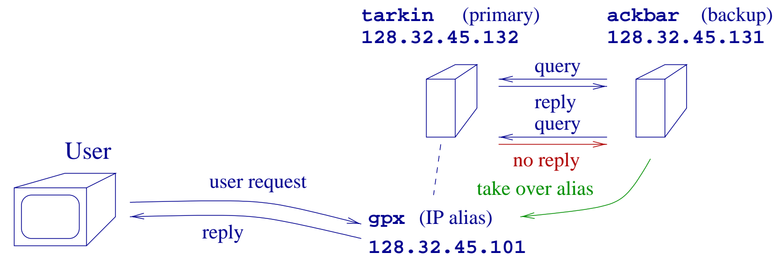

Figure 5: Masking failures

6.3.1 SCSI/Disk Subsystems. The SCSI and disk subsystems were the ones that caused most problems for us. It is understandable because commodity hardware are not necessarily designed with a large server system like ours in mind. This is one place where specialized storage system manufacturers have an advantage. We have been able to fix several problems by summoning help from FreeBSD developers. The actual nature of the problems are beyond the scope of this paper; please see our home page for more information.

6.3.2 Disk Identification. One problem with having several hundred disk drives, all looking identical, is that it is very easy to confuse them. SCSI disks are identified by their SCSI IDs within their bus, which are identified by the host adapter number within the machine, which are identified by hostnames and IP addresses. The problem is that if two disks are accidentally exchanged, the operating system will not be able to tell the difference; it will try using the disks until something falls over. The SCSI IDs are set on the disk enclosures, so if a pair of disks are accidentally swapped, for instance during replacement of a broken enclosure, their IDs will change with the disk in the same location appearing to having the same ID as before. There could be several different consequences ranging from OS crash to application error; some of them are extremely dangerous.

We are designing a system in which the system will be able to tell when disks are installed incorrectly. There are various ways to implement this, with trade-offs in the effects and complexity of design. Here the few alternatives we have considered.

Serial Numbers Each disk has a unique serial number in its permanent memory. By reading this, the operating system will be aware of disk being moved around by keeping a list of serial numbers of the disks and comparing disks to it upon each boot.

Comment: It is easy to implement, but will not let us do more than simple safe-guarding against operator errors.

Disklabel Each disk has a "disklabel" which describes the partitioning of the disks as well as some other information about the disk (rotational latency, geometry, etc.)[26]. It should be possible to expand the format of the disklabel to include some more information, such as expected bus/SCSI IDs and mountpoints.

Comment: We can do more than the previous option; it can be used for simple safe-guarding with bus/SCSI IDs (note it doesn't even require a table to be kept on the system, as the disks "know" where they are supposed to appear on the system) to more complex tasks such as automatic mounting. However, since the disklabel has a fixed size, there is a limit on the complexity of what we can do. For instance, we'll need to add special fields to describe if the disk is part of a striped set, and if it is, how many other disks are there, where in the set this disk appears, etc.

Script The last option is for each disk to have a "known" partition on which there is a filesystem where there exists a script that is to be executed in turn when the system boots.

Comment: What we can do with this option is virtually unlimited. It can be used to check the disk's identity (the system boot process should call the disk's script with its bus/SCSI IDs as arguments), it can be used to auto-mount necessary filesystems, or it can be used to construct more complicated entities such as striped arrays. Note that in either case, there is no modification whatsoever required on the boot disks themselves; the boot disks just read in the scripts and execute them in turn. If the scripts are

written cleverly, it may even not matter what order the disks are inserted in a particular enclosure. If a disk is moved to a different machine, it is not possible for our system to rectify the situation without an aid of an operator; we will need some kind of distributed filesystem to handle such cases.

We have implemented the third option, the script method. It is possible to move disks around within the same machine and still have them mounted correctly, both for the single filesystem case and striped array case.

6.3.3 Fast `fsck`.

It is important to reduce the time required for rebooting the system in order to minimize the window of vulnerability. We have implemented some methods outlined in Mary Baker's Ph.D. thesis[23], but the largest amount of time taken on a reboot has always been the **`fsck`**[27] time. It takes about 20 minutes to run on our 15-disk striped arrays after a system crash.

Dr. McKusick, the author of **`fsck`**, has been working on a project called "soft updates", in which by changing the way dirty data is written to the BSD filesystem, the performance and reliability improves greatly. These filesystems also do not need **`fsck`** to be run even after a crash, as it may lose some space, but they will be consistent[28]. We still need to run **`fsck`** from time to time to reclaim the space, but it can be done at any time, not right after a crash.

There has been snapshots of soft updates being released for FreeBSD. We have been using it on our machines for a few months with no ill effects.

7 Validation

We are planning to run simulations to validate the feasibility of our approach. The objective is to collect enough data on failures and prove by simulation that the system will actually run continuously and flawlessly in the existence of failures. We would like to be able to predict the expected downtime depending on several parameters, most notably the repair interval.

7.1 Collecting data

We have been keeping a log of all the failures we experienced. Many of them have distinct entries in system logs (which I have modified the system to keep for much longer than the default). In addition to observing the components under normal use, we are subjecting some disks to artificial loads to see if it will make any difference in failure rates. This in part augments the low loads we're seeing on the Museum project.

7.1.1 Logs.

There are various logs in the system that can be used to observe component failures. The main system log (**`/var/log/messages`**) is where all the kernel messages, as well as messages from any process using the **`syslog`** facility, go. Disk failures show up here as a loud and continuous stream of retries and failures. You can also see processes getting killed due to various reasons. Most of them are segmentation faults due to programming bugs, while there are some of them, such as swap pager faults, are caused by hardware problems.

The HTTP servers have their own logs. There is one file for access logs (**`httpd-access.log`**) and one file for error logs (**`httpd-error.log`**). The latter can be used to determine when the servers are restarted, etc.

7.1.2 Error frequencies.

The frequency of errors we have seen so far in 20 months of operation is shown below. The failures do not include components that were already bad as installed. Components without any failures, such as CPU and enclosure fans, are not shown.

Component	Total	Failed	%
SCSI adapter	44	1	2.3
SCSI cable	39	1	2.6
SCSI disk	396	7	1.8
IDE disk	24	6	25.0
Enclosure (SCSI)	46	13	28.3
Enclosure (power)	92	3	3.3
Ethernet adapter	20	1	5.0
Ethernet switch	2	1	50.0
Ethernet cable	42	1	2.4

Here are some observations:

- The Ethernet switch obviously has too small a sample size to draw any meaningful conclusions.

- On the other hand, the disk enclosures' SCSI bus integrity and IDE hard drives are major causes of concern. These are also the two hardest components to replace, causing a major headache for the system administrators. We are investigating methods to boot the machines with CD-ROMs as system disks, to avoid the problem with IDE hard disks altogether.

  One thing to note is that it is very hard to diagnose SCSI bus integrity problems. It can be any of the SCSI host bus adapter, cable, disks, enclosures or the terminator. As can be seen from the table above, most of our problems have been due to enclosures.

- Compared to the IDE drives, SCSI drives have been surprisingly reliable, the MTBF works out to about 400,000 hours. We suspect the difference is due to two factors: IDE drives being of lower quality in general, and the superior cooling of external disk enclosures that house the SCSI drives.

7.2 Simulation

We are planning to write an event-driven simulator to use the data we collected to experiment with various design parameters. The design parameters include: different repair intervals, systems with and without double-ending, systems with and without fast reboot optimizations, different disk failure rates, and different enclosure failure rates. It will also allow us to investigate radically different designs, such as having significantly more disks per machine.

8 Scaling to the next level

Our machine room, 10 meters on a side, has enough space to hold about 10 of our 400-disk systems. Using the recently introduced 50GB 3.5 inch drives such as the Seagate Barracuda 50, that will give us a total capacity of 200TB. That system will be able to hold 150 million GRIDPIX images or 15 million TIFF images of similar quality as ours.

Will our design scale well past the current size given the nature of the application, a web-based image server? We believe the answer is yes. Since each machine has its own web server, there is no real performance bottleneck except for the frontends and network connections to the outside world, as long as we keep the PC-to-disk ratio constant.

8.1 Performance

Our system's performance is already limited by the external bandwidth. Using a synthetic workload, we've measured that one of our FreeBSD system with 32 disks can sustain 2,000 8KB random reads per second, which translates to about 16MB/sec or 128 Megabits/sec. (With 1MB random reads, the number goes up even higher, to about 70 MB/sec.) Compared to the internal network bandwidth of 100Mbps switched Ethernet or external bandwidth of an 155Mbps ATM link, it is clearly impossible for the users across the Internet to generate loads anywhere near the maximum workload that a few dozen PCs can handle unless the load is extremely unbalanced. Having more machines in the system is not going to make the situation change.

The frontends will not be a bottleneck either. The loads on the frontends are extremely low—the request from the user passes through the frontend only once per image, when the user is first transferred from the museum, and the rest of the transactions, including all the **`mkhtml`** and **`gettile`** calls are handled by the backend servers. What the frontend does is a simple table lookup to determine the right server. Since the images are organized by their original PhotoCDs, each of which contain about 100 images, the size of the table is small too (only 800 lines so far—there are only 800 PhotoCDs).

We expect the frontends to handle the load of ten times as much servers easily. However, if the situation changes it is easy to add more frontends and have them work in parallel.

All of this, of course, depends a lot on the nature of the application. Our application does not require much processing power to handle user requests. Part of it is the nature of being the backend of an image database—the museum site handles all the database queries. Another reason is the design of the application. As mentioned in section 4, the GRIDPIX server requires very little processing power. For a more CPU-intensive application, particularly those that require more internal communication, might not scale nearly as well.

8.2 Maintenance overhead

Given that the performance scales well into the next order of magnitude in terms of number of disks, the cost of maintenance will be the main problem. Our system started out with one system administrator spending about 4 hours per day to keep it running; with the improvements we have made so far, it is now down to about 1 hour per day. When the self-maintaining system is fully in place, we expect this to come down to a couple of hours per week. With that workload, we believe it will be fully possible for one operator to handle a system ten times the size of ours easily.

The other aspect is the simplicity and modularity of our tools. Our monitoring is done by small, simple programs, running locally on each node, so adding more nodes will have little effect on their functionality. Also, by having pairs of machines back up each other as the first level of redundancy, there is not much additional overhead associated with increasing the number of nodes.

9 Summary

We built a 3TB storage system using only commodity components and have been using it as a web-based image

database for a few months. Despite frequent component failures, we have been able to give the illusion to users over the Internet that the system has never failed. We are improving the system to reduce the system administration cost. We believe our design scales well into larger sizes.

Acknowledgments

We would like to thank Bob Futernick, Dakin Hart and Sue Grinols of the Fine Arts Museums of San Francisco for photographing and cataloging the images and making this project possible.

This research was funded by DARPA Roboline grant N00600-93-K-2481 and the State of California MICRO program. We also thank IBM and Intel for donating disk drives and PCs, respectively.

References

[1] Strategic Research Corp. *Network Buyer's Guide*. http://www.sresearch.com/.

[2] John Wilkes. Private communication.

[3] Joel Bartlett, Wendy Bartlett, Richard Carr, Dave Garcia, Jim Gray, Robert Horst, Robert Jardine, Dan Lenoski, and Dix McGuire. Fault tolerance in tandem computer systems. Technical report, Tandem Computers, Inc., 1990.

[4] Network Appliance, Inc. http://www.netapp.com/.

[5] Nisha Talagala, Satoshi Asami, Tom Anderson, and David Patterson. Large scale distributed storage. Technical report, UC Berkeley, 1998.

[6] Fine Arts Museums of San Francisco. *The Art ImageBase*. http://www.thinker.org/imagebase/.

[7] Eric Anderson and Dave Patterson. Extensible, scalable monitoring for clusters of computers. In *Proceedings of the 11th Systems Administration Conference (LISA '97)*, pages 9–16, October 1997.

[8] Joel F. Bartlett. A nonstop kernel. In *ACM Symposium on Operating Systems Principles*, 1981.

[9] Tandem Computers white paper. NonStop availability for NonStop Himalaya and Windows NT server systems. Technical report, 1997.

[10] Tandem Computers White Paper. Tandem maintenance and diagnostic system (TMDS). Technical report, 1997.

[11] William J. Bolosky et al. The tiger video fileserver. In *Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 96)*, April 1996.

[12] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the tiger video fileserver. In *ACM Symposium on Operating Systems Principles*, pages 212–223, 1997.

[13] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996.

[14] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. Technical report, University of Wisconsin, June 1989.

[15] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[16] Garth A. Gibson, David F. Nagle et. al. A case for network-attached secure disks. Technical report, Carnegie-Mellon University, 1996.

[17] Tom Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randy Wang. Serverless network file systems. In *ACM Symposium on Operating Systems Principles*, 1995.

[18] Margo Seltzer, Keith Bostic, Marshall M. McKusick, and Carl Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the 1993 Winter USENIX*, pages 119–130, June 1985.

[19] Thomas E. Anderson, David Culler, David Patterson, and the NOW Team. A case for networks of workstations. *IEEE Micro*, pages 54–64, February 1995.

[20] Elizabeth Borowsky, Richard Golding, Arif Merchant, Elizabeth Shriver, Mirjana Spasojevic, and John Wilkes. Eliminating storage headaches through self-management. In *Proceedings of the 1996 OSDI*, October 1996.

[21] Sun Microsystems (white paper) Jim Waldo. Jini architecture overview. Technical report, 1998.

[22] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9), September 1991.

[23] Mary Baker. Fast crash recovery in distributed file systems (Ph.D. thesis). Technical report, UC Berkeley, 1993.

[24] Satoshi Asami. *GridPix: the Interactive Tile-based Image Viewer*. Technical report manuscript, available from `http://now.cs.berkeley.edu/Td/GridPix/`.

[25] Nisha Talagala, Satoshi Asami, David Patterson, Bob Futernick, and Dakin Hart. The Berkeley-San Francisco Fine Arts Database. In *Proceedings of the Fifteenth IEEE Symposium on Mass Storage Systems*, March 1998.

[26] Disklabel - read and write disk pack label. In *4.4 BSD System Manager's Manual*. O'Reilly & Associates, 1994.

[27] Marshall Kirk McKusick. Fsck - The UNIX File System Check Program. In *4.4 BSD System Manager's Manual*. O'Reilly & Associates, 1994.

[28] Marshall Kirk McKusick. Private communication.