112

Availability in the Echo File System

Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann

Friday, September 10, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Availability in the Echo File System

Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann

The Echo file system project explored several issues in the design and implementation of distributed file systems. This paper describes the aspects of the Echo design that are related to providing high availability. These aspects include the provision of redundant components (replicated disks and backup servers), the replication of information, and recovery from failures. Further, we discuss some less obvious mechanisms needed for providing truly high availability: load control, dynamic reconfiguration of the system, and the detection and reporting of faults. Finally, we discuss some of the impact of our availability mechanisms on application software.

CONTENTS

1. INTRODUCTION

The Echo file system project had several goals. We explored the use of global naming for all files in a large distributed system; we explored the provision of strong data consistency guarantees in a distributed file system with client caching, and we explored techniques for providing a high availability filestore in a distributed system. In this paper we consider only the high availability aspects of the project; the other topics are described in companion papers [1, 4, 8].

The basic technological components needed for a high availability file system are quite well known and understood. A naive designer might think that all he needed to do would be to replicate all the data storage and access paths, arrange for fail-over when a replica breaks and for recovery when a replica is repaired. Then do a competent job of the implementation and everything would be wonderful. The arithmetic looks good: modern disks have mean time to failure of 150,000 hours or better; modern computers have mean time to failure of at least several months. With bug-free software, the entire system would appear to its users to be almost perfectly reliable.

The reality is somewhat harsher. Producing a truly high availability system requires detailed and careful attention to numerous aspects of the design, implementation and management of the system. Several of the design issues were much more complex than we initially assumed, and there are more of them than you might think. We believe the results of our design and experimentation have taught us how to achieve high availability. This paper is written to describe what we learned along the way, in the hope that future travellers will find the exploration easier.

If you view personal distributed computing from the bottom up, it's easy to see the attractions of providing high availability. When we give each user a personal computer (PC or workstation), the user becomes autonomous. Regardless of administrative or mechanical failures elsewhere, the user can do useful work as long as their personal computer is up and running. And these personal computers are sufficiently reliable that they are almost always up and running. Then we notice that users often need to share files, or we notice that users need access to more file storage than can economically be attached to a single workstation. So we add to the distributed system shared file servers. But suddenly the user has lost autonomy and reliability. The failure of a computer that he's never heard of can stop him from getting his work done [7].

You can also view a distributed system in a top-down manner. There are many advantages to building a large system out of small components. But by using multiple computers to provide what the user thinks of as a single service, we have increased the likelihood that a necessary component will fail and make the service unavailable (since there are more necessary components). So we must compensate for this new source of failure by using still more components, in such a manner that when one fails others will take over the relevant tasks. By doing this well, we can build a distributed service that is at least as reliable as the user's workstation. If we succeed, the resulting distributed system will have the good aspects of single machine systems (shared resources and shared administration), with the added advantages of distributed systems (expandability, autonomy, scalability, economy).

The remainder of this paper proceeds as follows. First, we give a brief overview of the Echo design, just enough to understand the discussions of availability. Then we describe the relevant constraints that we adopted in our design, including a more precise description of what it means to provide “high availability”. The rest of the paper consists of a series of sections discussing in turn each of the mechanisms that are required to achieve system-wide high availability, namely:

- Section 4: eliminating single points of failure by introducing redundant components, so that when one fails we have an alternative component ready to go (e.g. backup servers and redundant communication channels).
- Section 5: replicating data so that any needed information will still be available after a failure (e.g. using mirrored disks).
- Section 6: using timeouts to detect failure. In particular, we use a technique that allows both parties to know that a timeout has occurred before either party acts on the timeout, even in the face of network partitions.
- Section 7: resynchronizing so that the replacement for a failed component can reconstruct the state of the original (e.g. agreeing on the state of operations that were outstanding at the time of the failure).
- Section 8: enforcing flow control policies to keep the load on the servers and clients low enough so that they do not run out of fixed resources or miss their availability deadlines.
- Section 9: providing for online reconfiguration so that servers and clients can keep running and providing service while the service is being reconfigured.
- Section 10: ensuring timely repair so that damaged components are replaced soon after they fail, reducing the chance that the next failure will cause the system to enter a state that is not available.

Finally, in sections 11-13, we discuss some of the impact that the Echo availability design has on application programs, and offer a few conclusions.

2. A BRIEF OVERVIEW OF ECHO

Echo is a client-server distributed file system. Files are named globally. In other words, the Echo name space is a hierarchy with a single root. To first approximation, each client of Echo sees the same file system name space.[1]

The top few levels of the Echo naming hierarchy are implemented by a distributed name service, and the lower levels are implemented by the Echo filestore.[2] The name service is not described in this paper, only the filestore. Briefly, the reason for using two different mechanisms for parts of the namespace is that the availability requirements are different. At the highest level, the update rate is low but the penalty for non-availability is enormous—if the global root isn't available, no files whatsoever can be accessed. So at the high levels of the namespace it's appropriate to use traditional name server algorithms, with their weak consistency guarantees but extremely good availability. But at the lower levels where users' real files are stored, we need much stronger consistency guarantees, and we are willing to pay for them with somewhat lower availability or somewhat higher cost.

We use the word *client* to mean a program (e.g. a Unix™ application process) that makes file system requests. We support multi-threaded software: a client may have multiple outstanding file system requests at a time. The client runs on its *client machine*. On the client machine, the file system requests made by clients are directed to a *clerk*, which is system code that is logically part of the Echo filestore, but which resides on the client machine. The clerk is responsible for all communication with the Echo file servers; the clerk knows which server is primary amongst replicated servers, and fails over to a new primary as appropriate. The clerk is responsible for multiplexing and de-multiplexing client requests, and for enforcing security distinctions between the clients. It also manages the caching and write-behind for the client machine.

The user's view of the Echo filestore is that it consists of *volumes*. Each volume is an arbitrary sub-tree of the global name space, designated as a volume by the system administrators. A volume has a unique parent in the namespace, and may have multiple child volumes. From the user's point of view, each volume is unique. The implementation might use replication to improve a volume's availability, but this is invisible to the client.

The Echo file servers group volumes into *boxes*. There is no necessary relationship amongst the volumes in a box—they can be from various parts of the global namespace. The file system administrator creates boxes from his physical disks, and chooses the availability characteristic of each box (e.g., whether the box is stored on mirrored disks). In other words, each volume is stored in some box, and each box is stored on some disks.

The Echo file servers are user-level processes running on computers that have access to the boxes (implemented on disks). The servers are stateful, and are capable of replicating their state. Servers are responsible for maintaining the integrity of the boxes

---

[1] This isn't strictly true—there are indeed some context-dependent names, but they are irrelevant to this paper.

[2] This is an over-simplification: the Echo name space is highly versatile, using an open-ended "junction" mechanism to join together "volumes" that are implemented by different services.

and volumes. All file system actions are ultimately performed by one or more of the Echo file servers, which commit the actions to stable storage.

If the administrator decides that the server for a box should be replicated for higher availability, the replicated servers organize themselves using a primary/backup algorithm. The servers cooperate amongst themselves to choose one of their number as a *primary server* for the box. Clerks communicate only with the primary server for each box. The primary server propagates requests as appropriate to *backup servers* for the box. Primary servers replicate much of their volatile state in their backup servers. If a primary server fails, one of its backup servers will become primary. This is all explained in greater detail later in this paper.

If the administrator decides that the contents of a box should be replicated in order to provide higher availability or reliability, the Echo file servers manage *box replicas* for that box. A *quorum* is a set containing a majority of the replicas of a given box. A box replica can either be a *full replica* or a *witness*. A witness stores just the information needed to determine whether an up-to-date quorum of box replicas has been formed, while a full replica stores a complete copy of the data in the box in addition to the quorum information. Replicated boxes typically have two full replicas and one witness; an unreplicated box has a single full replica. Box replicas may be added or deleted while the box remains in service without disrupting service, as long as a quorum of the remaining replicas is available and at least one of the remaining replicas is up to date.

Note that the concept of replicating a server is quite distinct from that of replicating a box's data. The two actions have different behaviors, different benefits and different costs.

Finally, a *disk* is the basic unit of stable storage used to build box replicas. A disk may be dual-ported to each of two file servers (primary and backup); or it may be single-ported to one of those file servers; or it may be accessible through a network disk server. While each of these three options can offer availability in the face of a single failure, each has different availability properties. For example, single-porting a disk to one of the file servers means that if that file server fails, then the box replicas stored on the directly connected disks also become unavailable. In this case the server connected to the disk can return to service soon after being rebooted but each of the box replicas will remain out of date until its state can be restored from an up-to-date box replica. In the configuration in use at SRC, the disks are all dual-ported, and each stores part of the contents of a full box replica plus possibly the contents of a small number of witness box replicas for other boxes. Note that we speak of Echo boxes being replicated; it's not useful to think of Echo disks themselves being replicated, since they must be aggregated into boxes before they have any useful semantics.

3. CONSTRAINTS

In this section we describe the constraints on the Echo design that affect availability. We're not concerned here with the trivial or obvious—the system should store files correctly, and should perform at a level commensurate with the hardware, software and administrative investments. But we made some decisions in the overall Echo project that affect the availability design; decisions that you could rationally argue for making differently. These we discuss here, together with a more exact description of what it takes to be able to claim “high availability”.

Consistency Requirements

In Echo we decided that our distributed file system should retain the property of global data consistency. More precisely, the semantics observable by Echo’s clients are those of single-copy serializability. All the observable behaviors of the Echo file system, in the absence of failures from which it does not claim to recover, are equivalent to some serialization of the multiple clients’ actions as if made to a single copy of the data.

Most other distributed file systems provide weaker consistency guarantees than this. For example, in NFS™, the world’s most popular Unix-based distributed file system, modified data can be kept in a client’s cache while a file is open, with the modifications being totally invisible to another client reading the same file. Multiple clients of NFS can simultaneously see different values for the file data.

Two other distributed file systems, Locus [11] and Coda [6] include provisions for replication, but take a strikingly different stance on consistency. In both of those systems clients can continue to operate even while they are out of touch with the servers (because of network partitions or server crashes). To permit such disconnected operation, both of those systems rely on reconciliation algorithms when a client, having made updates while disconnected, once more contacts the file servers. Such algorithms necessarily have failure cases (which require human intervention). In effect, Locus and Coda trade-off weaker consistency guarantees to achieve higher availability.

Finally, there are many examples of distributed, replicated data storage services that provide entirely satisfactory semantics without single-copy serializability. Examples include the various name services, 1-safe databases, and repositories used for software version control systems. In each of these cases, application-level considerations make single-copy serializability unnecessary, and the resulting freedom makes for better implementations.

We believe that it is important for a general-purpose file system to strive to continue to provide the single-copy model to which almost every application was written. Providing single-copy serializability is the only way to be sure that the distributed file system will give the same answers to the same questions as a centralized file system would have. A system that nearly always provides single-copy semantics does the user no favors—the times when the file system violates single-copy semantics will be rare, unpredictable and hard to reproduce. Applications that use such systems without having been specially customized to cope with their peculiar semantics will inevitably be unreliable and untrustworthy.

Safety Requirements

The *safety* properties of a system are a description of what the effects of the system might be, with no guarantee about how soon, if ever, the effects will happen. It is useful to distinguish safety properties from other properties, such as liveness (whether the system will reach the specified states), performance (how soon it will do so) and availability (considered in the next sub-section).

A system is *fail-stop* if it will never give an incorrect answer (although it might return an error indication, or it might compute forever). A system is *byzantine* if it isn't fail-stop. In the Echo design, we assume that clients, including their clerks, are byzantine. We assume that the servers' hardware, the disks and the network (software and hardware) are fail-stop. We intend that the Echo file servers will be fail-stop. To achieve this we rely on the following conditions:

- The overall fail-stop behavior of the Echo servers and of other Echo clients is unaffected by a misbehaving client, including a misbehaving clerk. At worst, as misbehaving clerk can cause clients on that clerk's machine to get incorrect answers, and might cause performance or liveness problems for the servers or for other client machines, but it will not cause the servers or other client machines to get incorrect answers.

- The servers will give the correct answer to all requests from clients or other servers, or will give an error indication, or will give no answer at all. Of course, in reality server software and hardware do not always satisfy this: sometimes there bugs that cause the wrong answer to appear. What we are saying here is that if the servers are actually fail-stop, clients will get the correct answers; but if servers give incorrect answers, clients will in all likelihood get incorrect answers. On the positive side, we are specifying that even if our servers have arbitrary performance problems, including arbitrary unfairness in the thread scheduler, any answers that they deliver will be correct answers.

- Our algorithms assume that the disk hardware is fail-stop, in the following sense. If a server initiates a write request, the request will modify at most the specified part of the disk. If a server issues a read request, the request will return an error, or will return correctly the data provided in the last write request for that part of the disk.

- Our algorithms assume that our network communications are fail-stop. We use communication protocols that provide error-free, duplicate-free, in-order messages (actually, remote procedure calls). We assume that these semantics will be satisfied, or an error will be reported by the communications sub-system. Of course, the underlying communications hardware has weaker properties, but our RPC system provides these additional guarantees. When an error is reported to a process that has sent a message, the message might or might not actually be received by its intended recipient (i.e., there might be orphans). Note that our algorithms are correct even in the face of network partitions, or non-transitive network service interruptions (e.g. A can talk with B and B with C, but A cannot talk with C).

- In addition, we assume that there is a known bound on clock drift amongst the participating machines. In other words, there is a known universal constant M such that when two machines measure an elapsed time interval at least as long as N, the ratio of the interval length measure by one machine to that measured by the other machine will never exceed M. Note that this is a constraint on elapsed time measurements, not time-of-day measurements. For our environment we know M=1.2 and N=0.4. The reason that M is so large is because our time-of-day clock synchronization algorithm allows clocks to run 10% fast or slow to adjust to the correct time-of-day, and we use the same clock for elapsed time measurement. We should have arranged for the underlying system to provide a separate elapsed time value, which could have had much greater accuracy, but we did not do so (for reasons irrelevant to this paper).

If any of these assumptions is ever violated then the system makes no guarantees on its correctness either now or in the future. In practice, a violation can cause some number of boxes to get into a bad state, perhaps a state so bad that the server crashes while examining it. In this case the disks storing the box have to be reformatted, a new box created, and the old information restored from an offline archive.

We have had two known instances while in service where software errors caused our servers to become byzantine and break our semantics. In one case the bug resulted in two servers both becoming primary when each server actually owned one real replica and each thought it owned the witness. The two replicas then diverged as each server made changes to its disk. In the second case, a single server got confused and thought both replicas were up to date when one was in fact out of date. In both cases the damage was limited: we were able to recover the data from the online replicas and did not have to restore from older offline backup tapes.

We have had several instances where undetected hardware faults on our experimental multi-processor caused strange behavior. For example, we had several cases where a memory fault caused a bit to flip in main memory without being detected by the memory's parity check circuits. Many of these hardware faults were detected by the use of various defensive programming techniques in our software, while others were reflected in bits being corrupted in a user's data. Luckily no such fault has caused large scale loss of data.

We have never encountered byzantine behavior in our disks or communications subsystems.

Availability Requirements

A naive notion of availability is that the system is available if it is always able to respond to client requests. But this is much too simplistic. For example, responding to one client but ignoring others does not provide satisfactory availability. Equally, responding to all clients but so slowly that the user has given up before the response arrives is not effective availability. The truth is that there is little distinction between an availability specification and a performance specification.

So we need to be more explicit about what the Echo project aimed to provide by way of availability. Here are three different potential definitions of availability:

- *Liveness:* A service is available if it can be shown that every request to the service will eventually complete.

- *Throughput:* A service is available if the service can complete K requests in every X-second interval, for satisfactory values of K and X.

- *Real-time:* A service is available if every request completes within X seconds of being issued, for a specified value of X.

For the service to meet any of these definitions of availability, the environment will have to satisfy additional requirements beyond those required for safety. These requirements get more stringent as we go from liveness to throughput to real-time availability.

Liveness and the concept "eventually" can be made mathematically precise using temporal logic, but they are not properties that can be contradicted by measuring the system—you cannot tell by observation that a request will never return.

Real time availability imposes stronger requirements on the environment because, given limited computing resources and a fixed cost of performing an operation, the offered load on the system must be limited if a real time requirement is to be met.

Throughput availability can be achieved even if the offered load on the system is unlimited, as the server is free to tell its clerks to defer requests and the clerk is free to block its clients. But we must still bound the load on the servers by restricting the number of new machines coming online.

A further complexity in defining real time or throughput availability is that the load caused by a particular file system request depends not only on the request itself but on the state of the service when the request is given. For example, a request to write one byte of a file might be issued while that file is cached by hundreds of machines. In this case updating that byte would require communicating with each of these machines to invalidate their caches. A client wishing deterministic response times must know enough about the structure and usage of the service to predict the load caused by an operation. This is a general problem when complexity and optimization for the normal case interact with real time deadlines.

Echo attempts to achieve high availability with guaranteed throughput. To do so, we use redundant components, replicated data, mutual timeouts and load control. All of these are discussed in later sections of this paper.

4. REDUNDANT COMPONENTS

If a service is to be highly available, each component in the service must be either redundant, or very reliable.[3] In Echo we added redundancy at many places in the system. In this section we identify the redundant system components and discuss the structural implications. In later sections we discuss the more detailed issues relating to making redundancy actually work—replication of information, detection of and recovery from failure, and so forth.

Notice that in Echo we chose not to enhance the availability of the client machine itself. If a client machine fails, all the programs running on that machine fail and there is no automatic mechanism for restarting or reconstructing their state. If a user is writing a highly available application or service, intended to continue working after the failure of a machine it is running on, then the user has to write the application specially. Echo does provide facilities that make this easier than it would be on an unreplicated file system (e.g., global data consistency and global file names). Some techniques for writing highly available applications on top of Echo are described in a later section.

Network Connections

High availability remote file service requires high availability network connections. There are several ways of achieving this. For example, you could run parallel Ethernet cables, and give each machine two Ethernet controllers. In our environment we had a more effective solution available, in the form of an experimental mesh-connected switch-based network known as AN1 (formerly Autonet) [12]. The AN1 is self-configuring, and includes the ability to run redundant links. This redundancy extends to having two network cables into each connected machine. When one link fails, the network automatically reconfigures to use another route. In AN1 this reconfiguration happens in about 100 milliseconds. This behavior is sufficiently good that Echo can meet its availability goals by considering the local area network to be perfectly reliable. (Packets are still lost occasionally, of course, but this is covered up by retransmission algorithms within our RPC transport protocol.)

The worst case that we might realistically encounter is the failure of a machine's network adaptor, which we did not replicate. This event should probably be considered as a failure of the machine itself.

---

[3]A component that can fail and recover very quickly, that is, within the bounds of the availability timing requirement, can be considered to be very reliable.

During the two years that Echo has been in service we have had many failures of the Ethernet in our building, but we have had only two failures of the AN1, both of them quite short and both due to software bugs.

The management of AN1 to ensure that there is sufficient redundancy isn't really a part of the Echo project, although it is certainly one of the management tasks needed to achieve genuine high availability.

Disks

Modern disks have much lower rates of failures than processors. Nevertheless redundant disks are important:

- Even if the disk itself doesn't fail, all of the paths to the disk might fail, resulting in the effective failure of the disk. This is more likely for single-ported disks.
- The failure of a disk holding the only copy of important information can entail loss of the data, not just the temporary loss of availability; so although it is unlikely, its effects are so catastrophic that it is worth protecting against. An alternative in this case is to rely on off-line back-ups such as tapes; the issue here is how much are you willing to pay to avoid losing the data created since the last back-up?
- Even if a disk has its data preserved at a different site or on offline storage and thus can be recreated, the loss of availability while a restore is in progress might be unacceptable.
- Administering a system may require moving a disk physically and plugging it into a different circuit or host. Without replication this cannot be done without loss of availability.

The Echo system is set up to permit replicating the contents of a box (i.e., of a set of file system volumes). If the purpose of this is to increase availability (rather than just to preserve the data) then the replicas should be on separate disks with disjoint access paths. For example, the replicas should not both use disks on the same SCSI chain or be connected to the same host. The techniques that we used to replicate information on disks are covered in a later section.

In our installation we configured each Echo box to have two full replicas. This increases the cost of the system significantly, but certainly improves availability. Over the last two years we have had five cases where disk units have failed and in each case we had no loss of information and no loss of availability.

As a side note, the fact that all boxes were replicated let us take several short-cuts in the implementation:

- We avoided writing a file system scavenger that tries to recover data and re-establish the invariants after the loss of an arbitrary portion of the disk storing the file system, since instead we just copy the data from the other replica. It's possible to write a scavenger for the Echo disk structures, although our use of logging (described later) makes it somewhat more difficult—the loss of a page in the log can affect the value of any page on the system.

- Having two online replicas allowed us to feel comfortable with making file system backups only once per week. This was infrequent enough so that we could afford to always do a full backup, so it saved us the work of writing and testing an incremental backup system.

File Servers

The file servers are the most failure-prone component in the system, apart from client programs and their machines (whose failures have limited scope). Servers can fail because of software bugs or hardware failures. They can also effectively fail because of excessive load—this can make the server fail to perform work for a client within the real-time guarantee, or might make the server not respond in a timely manner to keep-alives and thus provoke a timeout. Finally, servers fail because of administrative activities such as software upgrades and hardware reconfiguration. For all of these reasons, the Echo design permits replication of servers. That is, we permit the administrator to configure the system so that more than one server is capable of offering service for each box. In our installation, we configured Echo so that every box had replicated servers.

When you replicate a server to increase availability, it must be the case in general that when one server fails, the replacement server will not also fail. In other words, the failures of a server and its replacement must be independent. Early on in the Echo project, when we were still flushing out the most obvious software bugs, we had quite a few bugs that were not independent—a particular client request would trigger a bug that caused one server to fail; then after its replacement took over, the same request would be resubmitted and the same bug on the replacement server would cause it too to fail. This double failure of course caused a loss of availability to the clients. We have found that the remaining bugs in our server code arise during relatively infrequent combinations of operations, and that in those cases the non-determinism inherent in aspects of our programming model (lightweight threads and fine grain locking), and the asynchrony inherent in some of our algorithms, cause operations to execute differently when re-executed on the replacement server.

Echo file servers (as distinct from the boxes they serve) retain no stable state specific to an individual server. Their function is to gather the resources needed to service a box (actually, a box's replicas), and offer service when a server has acquired a majority of the replicas of the box at least one of which is up to date. There is no voting or consensus protocol that has to be executed between servers—requiring that the primary own a majority of the disk replicas is sufficient to ensure that there is only one primary.

In Echo a single primary server performs all updates and reads for a box. Using a single primary for all updates was the only way we knew of achieving all of the following:

- Short client timeouts on update requests to the server. We don't have to worry about what happens when two servers attempt to work on the same operation, because at most one server thinks it is primary at any time—the other will discard the operation. So if a request is taking too long, the client can guess that the primary has died and give the request to the backup without needing to be certain that the first server is indeed dead.

- Strict consistency among box replicas. Having only one server means that that server can have complete information about the differences between the box replicas.

- No extra messages for each operation. The only messages required are parallel messages from the initiating server to each disk and the reply from each disk. Extra messages are required only when the set of replicas changes.

- Caching inside the server. We don't have to worry about another server doing an update that might invalidate the cache.

On the other hand, care is required to ensure that backup servers can tolerate the extra load that they will receive if some primary server fails. Reasonable economy dictates that each server should be primary for at least one box; thus when it takes over from some failed server, it must be able to service that load in addition to its own normal load. This is really an administrative trade-off: it's simpler to run the system if there are a few very large boxes, but this necessarily stresses the backup servers when a primary fails. In our installation, we opted for administrative simplicity in our local configuration: we had one box for the entire set of disks served by a server in normal operation. This meant that when a server failed, its backup would receive up to twice its normal load. In retrospect, this was probably a bad configuration decision; our implementation would have permitted us to use smaller boxes, and we should have.

Another way of balancing load is to allow non-primary servers to perform reads. This makes sense if non-primary servers can read from disks, which can be arranged except when using certain types of dual-ported hardware. We did not do this for several reasons:

- Our original expectation was that we would use large enough client caches so that at the servers there would be vastly more writes than reads. This has not turned out to be the case, at least for our work loads and our client cache size of 16 megabytes. In our current environment there are more reads then writes. We still find this fact surprising.

- It would have made the system more complicated—the primary and backup servers would need a cache coherence protocol to deal with writes done on the primary to data cached on the backup.

It seems likely that a better strategy would be to implement a *cache server* that acts as an intermediary, caching data on behalf of clients. Because a cache server is a shared resource and has no other load, you could afford to use several times the memory that you could assign to file caching on a workstation. We have not implemented cache servers, but they seem attractive: on our file servers, with 80 megabyte caches, we see a hit ratio of 60%. This indicates that a large cache server would be able to field at least half of the clients' reads; this would make it a more cost-effective performance enhancement than permitting reads at non-primary servers for the box.

Paths between Disks and Servers

In some configurations, replicating the disks and the servers isn't enough. You must also ensure that there is a communication from at least one copy of each box to at least one server for that box. Echo supports four ways of achieving this:

1. *Dual-Porting:* Disks can be dual-ported to two file servers and accessed directly. This takes advantage of hardware that was standard on the class of disks that were available to us during the life of the project.

2. *Pass-through:* Disks can be plugged directly into one file server and made available over the network to other file servers.

3. *Disk Server:* Disks can be plugged into dedicated disk servers and accessed over the network by multiple file servers.

4. *Dual-Ported Disk Server:* Disks can be dual-ported to two dedicated disk servers and accessed over the network by file servers.

The choice of which connection strategy to use is often dictated by cost, hardware capabilities and/or availability requirements.

The pass-through approach uses the least amount of extra hardware, but the disk is available only if the file server attached to the disk is up. The pass-through approach can be used to avoid single points of failure only when combined with replicated disks: the pass-through is the way that the primary server accesses the disks storing one of the box replicas. The other box replica would be stored locally.

The disk server approach consumes extra hardware and can add latency to disk accesses, but disk servers can have several useful benefits:

- They can implement a very simple protocol, requiring only relatively simple software. It's probably simpler to achieve satisfactory performance in the disk server than in the rest of the file service. Both effects can allow disk servers to be remarkably reliable.

- They allow for more flexible configuration options, as file servers need not be directly connected to any of the disks they are using.
- They ease the cabling problems that arise when you have many servers and disks whose connection pattern may need to change. If load balancing of disks between disk servers is not critical, each disk can simply be plugged into the closest disk server. Simplified cabling is one of the primary reasons for the popularity of Digital's HSC's among its large customers.

Note that because we are not willing to trust client machines, the clients cannot be permitted to issue commands to the disk servers directly. To enforce this, disk servers must be on a separate network, or must employ some type of authentication to ensure that clients cannot issue commands or otherwise break the file system's security guarantees.

In the configuration in use at SRC we dual-port each disk to two file servers. Managing many dual-ported connections has been a problem, as we now have two cables from each disk—one leading to the primary and one to the backup file server. Sometimes these cables can end up being quite long. Our systems, built out of Digital's 1985 vintage Q-Bus workstations, also have a tight limit on the number of direct disk connections that can be made to a single machine. For single-ported unreplicated disks this restriction is not too onerous, but for dual-ported disks—which require twice as many connections per host—this can be a problem, especially when disks are replicated (which doubles the number of connections again).

On our hardware, filling the file servers with the maximum quota of disk controllers has caused us to load the power supplies of our machines to the very limit of their specification, so much so that we have had several cases where the power supply wiring burned up. We now do periodic sniff tests to detect any burning wires on the servers!

Fortunately, supporting all the different configuration options doesn't add much to the complexity of the system. The file server implementation uses an object-oriented interface to access disks. There are both local and remote implementations of the disk object class: a single remote implementation layered on remote procedure calls, and a separate local implementation for each class of disk. This approach has made handling all of the different disk configurations quite easy, adding little complexity to the programming of the system and allowing lots of flexibility.

At one point, due to an administrative mistake, we ran the service on a machine with the local port to a disk turned off and didn't notice any problem for several hours because the server accessed all of its disks remotely through the server connected to the other port. This illustrates that even if you can provide flexibility without adding to the program complexity, it can easily add to the difficulty in configuring and managing a system.

When using any of the connection strategies you must be careful to ensure that disks have only one 'owner' at any given time. We want to avoid having writes from the old owner of a disk interfere with the new owner's reads and writes. Ownership of disks is also what we use to build the notion of ownership of a box replica, which in turn drives our primary determination algorithm. We implement ownership as follows. Each time a disk is opened, the server is returned a unique handle (from the disk driving software that is attached to the disk, either locally or remotely). Each disk operation takes a handle as an argument and performs the operation indicated, or fails with or without side effects, or fails due to having a bad handle with no side effects. At most a single handle is good at any one time and once a handle becomes bad it never becomes good again.

This mutual exclusion property is very easy to implement with a single-ported disk, because a single programmable host is in control of all access to the disk. With dual-ported disks, implementing the mutual exclusion property can be problematic given hardware limitations.

With some dual-ported disk hardware it can be difficult to achieve this mutual exclusion if a server is overloaded. For example, one thread on a server might attempt to issue a write to a disk, but before it can actually issue the write the server gets overloaded and is presumed dead by the rest of the system. This causes the backup server to open the disk, become primary, and start using the disk. There is then a risk that when the load on the first server decreases the thread doing the write on the first server might finally succeed in communicating with the disk and would actually perform its write, which would interfere with the operation of the new primary server. Even if the new server crashed in the mean time and the disk become free again, the state of the disk has changed and the original write request might well be the wrong thing to do.

Our implementation of dual-porting uses Digital's MSCP/SDI[4] dual-porting architecture. Each disk has two ports, which may be plugged into different disk controllers, generally attached to two separate hosts. Each disk has an arbiter which determines the single port by which the disk is owned. If the arbiter fails to hear from the owning controller over a certain period of time, the disk becomes available and either plugged-in controller can then become the owner of the disk. Each time the disk selects a port, the controller must first bring the disk online before it is allowed to issue operations to the disk, and the controller can bring the disk online only in response to a specific request by its host.

This combination of features allows us to implement ownership of SDI disks if we check the handle for validity before issuing the request, and wait until all operations left outstanding on a disk complete or fail before bringing a disk online again. This ensures that old operations from previous owners are never executed.

Implementing ownership on a dual hosted SCSI chain, one with ports zero and seven connected to hosts, is more difficult because the standard SCSI interface has no notion of 'online host' or long term locking. Either host can issue a disk command at any time. Typical dual-hosted SCSI schemes either use stronger processor failure models than we felt comfortable with or make modifications to standard SCSI.

---

[4]Mass Storage Control Protocol, Synchronous Data Interconnect, Trademark of Digital Equipment Corporation

Physical proximity and shared power supplies

Regardless of how many computers and how many disks you use, your availability will be poor if they all rely on a common power source. We have had limited goals in this area: most of our users work in a single building, and if some failure takes power away from most of the users' workstations, having the file service remain up would be of little benefit. Remaining available during a disaster, such as an earthquake or a widespread power problem, was thus not of concern to us. Recoverability (i.e. making it likely that we can get our services back up sometime), was an issue we did address (through a mix of off-line backup strategies) but that is not the subject of this paper.

However, we have occasionally (twice in two years) experienced a power interruption due to a tripped circuit breaker for our machine room. As part of the Echo project, we installed a second, separately tripped circuit for the machine room, and we split several of the cable trays so that the machines could have access to power from either circuit. We then plugged half of the servers into one circuit and the other half into the other, and arranged that the primary and backup server for each box were on different circuits. Disks were similarly split. To ensure availability, the witness disk replicas (those replicas that are used only to break ties when the full replicas are in network partitions) should be on a third circuit (e.g., out of the machine room altogether).

Another related area is ensuring timely return of service after a power failure is corrected. It's quite difficult to perform a cold-start of a system like Echo. The configuration information for the file servers is stored in our name servers, and the name servers store their data in files. To break this circularity with minimal code, we rely on our non-Echo computers: we run a name service replica there, and then use that to permit the Echo file servers to start up. Unfortunately, our non-Echo computers are quite slow and unreliable at starting, and this delay is added to Echo's own start-up delays, causing an overall poor recovery time (up to an hour).

A better way to break this circularity would be to store the configuration information for certain unreplicated file systems on the disks themselves. Then file servers could boot without a name service by using such a file system as the root of the file system tree rather than waiting for the name service to respond. The designated root file systems would be unreplicated and would contain everything needed for booting. Higher level management programs would keep the different root file systems up to date. Once the system had started, these root file systems would become accessible with a global name. We did not implement this scheme.

5. REPLICATION OF INFORMATION

There are two reasons for replicating information in Echo: some is replicated because loss of the information would result in a client-visible failure of the service (e.g., the contents of clients' directories and files), and some is replicated because it must be available before service can be provided after a failover and it would otherwise take too long to regenerate it. We discuss both sorts of replicated data in this section.

File system contents

In section 4 we explained why we want to replicate the file system contents: to prevent loss of data after a media failure, and to prevent poor availability that would occur while restoring from an off-line backup. Note also, as discussed in section 4, we do not try to use replicated data to improve performance by sharing load (for the reasons stated there).

The main difficulty in maintaining replicated data is that you cannot commit an update atomically onto two physical devices. All commit protocols have failure modes that cause the replicas to have different contents, and the designer must include algorithms to recover from this situation. To achieve this you must do two things. First, you must detect (efficiently) that a replica is out of date. Second, you must bring out-of-date replicas up to date. Without considering availability, these problems are easy (e.g., compare the disks and copy one to the other in its entirety). But high availability dictates more sophisticated measures. The out-of-date detection must be fast, and the operation of bringing a copy up to date should be able to run concurrently with providing service for new updates.

We have written a separate paper that described our algorithms for achieving atomic update of replicated data, and for detecting out-of-date replicas [3]; these algorithms are also summarized below. Our algorithm for bringing a replica back up to date while still providing service is described in a later section of this paper.

In order to keep track of which box replicas are up to date we maintain three *epoch* counters. These counters are maintained by each of the replicas, in stable storage belonging to the replica (actually, on the same disks). Additionally each replica maintains the identity of all the replicas, as a set of unique identifiers on stable storage. We use a special three-phase algorithm to advance these counters. This algorithm executes whenever the set of replicas serving a box might have lost a member. It runs on the server that expects to be the primary, a server machine that owns a majority of the box replicas, and fails if any replicas are lost while the reconfiguration is in progress. (When the algorithm “fails”, it stops running and is later restarted from the beginning.)

The epoch advance algorithm has the effect of marking all inaccessible replicas as being out of date, while no accessible replica is ever marked as being out of date. After the algorithm finishes we enter a *service period,* which lasts until the next failure. Each service period is labelled uniquely by the value of the epochs on the up-to-date replicas

The epoch counters are named *big*, *prospective*, and *service*. Because these counters continue to grow with each execution of the algorithm we allocate 64 bits for each, allowing for centuries of service. After two years of service, all of the counters in our configuration are under twenty thousand. Each replica also has a boolean called *dataOK* which is used to indicate that the data on the replica is up-to-date. This will allow out of date replicas to be brought back up to date while the box is in service.

The counter advance algorithm for box B is as follows. This algorithm is executed by any file server for B that thinks it should be primary for B. Note that if multiple servers execute the algorithm at most one of them will succeed. If they all fail, at least one of them tries again.

- Find one accessible replica A for box B.
- Obtain from A the set of identities of the replicas for B; call this set K.
- Attempt to acquire control of a subset R of those replicas, such that $|R| > |K|/2$ and each member of R has its service counter less than or equal to the service counter in A. If the server executing this algorithm can't acquire this majority within a suitable timeout period, it releases control of all replicas and starts over; this might permit some other server to become primary. The timeouts are arranged so that the timeout for building a majority is shorter than the timeout on waiting for an additional replica after getting a majority.[5]
- If, while collecting replicas, we find any replicas with a larger service epoch than the replica we have chosen, we start over using the replica set from the more up-to-date replica.
- Let S be the maximum service counter in the members of R.
- Let B be 1 + the maximum big counter in the members of R.
- If any member of R has a prospective counter less than S, clear the dataOK flag in that member. If dataOK is false for all replicas, then we start over. (In other words, if a replica missed the last service period, mark its data is being incorrect so that it can be updated later. If all of the replicas have bad data, fail and wait for a good replica to show up.)
- Set the big counter in each member of R to B. (This ensures that B will never be used again as an epoch.)
- Set the prospective counter in each member of R to B. (This marks each member of R as being part of service period B.) Also, store the set K in each member of R.
- Set the service counter in each member of R to B. (This has the effect that replicas not in R will be considered to be out of date in any future execution of this algorithm.)

During a service period, updates to the replicas are carefully managed by the primary server to make sure that the differences between up-to-date replicas are always well

---

[5] Our initial design used a preliminary server election phase, where the servers would elect one of their number to attempt to become primary before any would attempt to obtain leases from the replicas. This would eliminate the possibility of live-lock caused by servers fighting over replicas. But we didn't implement it because it was not needed in our environment.

defined and the differences can consist only of updates that have not been committed to the client. When service resumes after a failure, the in-progress updates have to be examined and either backed out or redone on all accessible replicas. We call this last process *reconciliation*.

One auxiliary issue in this area is the best semantic level at which to do replication. We considered three strategies:

- File system: Each replica has its own implementation of a file system volume. Because of the requirements of reconciliation, each replica must keep a record of file system operations that might not have been completed on some replicas, so that during reconciliation these operations can be backed out on this replica or applied to the other replicas.

- Array of disk blocks: A single implementation of a file system volume runs on the primary file server making updates to the disk replicas, treating each as an array of disk blocks. Each replica must keep a record of disk operations that might not have been completed on some replicas. Additionally, some higher level logging mechanism would be needed to complete or undo partially committed file system operations that involved multiple disk operations.

- Mini-transactions over an array of disk blocks: In addition to the array of disk blocks mentioned above, which we shall call the *home*, each replica maintains a log which records a collection of changes to the home. All updates to the home are first recorded on the log. For atomic operations the actual data is stored in the log; for non-atomic operations the log contains the list of the home pages being updated. Thus any differences between the two replicas are recorded in the log. To limit the scope of reconciliation, the difference between the lengths of the logs of two up-to-date replicas is limited. The total length of the log is limited by writing updated pages to the home location. When all of the updates written before a given point in the log have been applied to the home location, that portion of the log may be truncated.

The first strategy seems natural when each replica is single-ported to a server. It also allows the layout of each replica to be quite different. This provides a lot of flexibility, but it complicates free space management—the primary would have to call each of the replicas as part of reserving space for a client operation. It also increases the total amount of computation done. However the first strategy makes doing finer grain replication plausible as there is no relationship implied between replicated objects sharing the same pool of disk space.

The second strategy is more natural for use with dual-ported or network accessible disks, as the higher level processing to determine which blocks have to be accessed only has to be done once. Such strategies can be somewhat easier to program as we do not need to worry about logging high level information.

But on balance we decided that the third strategy was preferable. Each strategy requires a log of operations corresponding to file system primitives such as “rename”, in order to provide atomicity of those operations after recovery. Using the mini-transaction approach allowed us to use a single log for the higher level atomicity and also for recovery from raw disk operations in progress. We still believe that this is the correct decision. This issue is considered in detail in a separate paper [4].

Log Information

When a server becomes the primary for a box, it has collected a majority of the available replicas of the box’s data, and has ensured that at least one of those replicas is up to date (i.e. was a replica in the last service period). As described briefly above, and more extensively in a separate paper [4], our on-disk representation of the box consists of a set of home pages, modified by a log (or journal). When the new primary server starts up, it must read the log so that its dynamic data structures reflect those pages that have been committed to the log but not yet written to their home locations. This allows the server to deliver the correct values for those pages without searching the log.

The simplest approach would be to read the log sequentially from the beginning. Unfortunately, the time that this would take (about 20 seconds in our configurations) is time during which we cannot offer service for the box.

It would be possible to prevent this delay by replicating on the backup servers the dynamic state that would result from reading the log. This can be done by the primary server forwarding log entries to the backup servers asynchronously, after committing them to its own stable storage. Each backup server could then process the log just as it would have done had it read it off the disk. This processing can be done at a leisurely rate on the backups, provided that on average it can keep up with the primary’s throughput. If a backup server becomes primary after some failure, it needs to read the log entries from stable storage only to the extent that it was running behind in applying them. (During normal running, the log is write-only, since all the affected state is also in volatile memory on the primary.) We did not implement this part of the design.

We considered three proposals for this portion of the design:

- *Locations only:* The mechanism described above keeps track of both the locations and contents of dirty pages. An alternative is to keep track only of the locations of the dirty pages. When a dirty page is accessed the primary might have to read the log to get the latest value for that page. This reduces the amount of information that the backup has to process and store, but will increase the disk load on a new primary as it is forced to read the log.

- *Complete dirty pages:* We could have made sure that the log always contained the complete value of any dirty page. This would sometimes increase the size of the log and use more disk bandwidth, but it would allow reads of updated pages to be satisfied without reading the home locations on disk; this could significantly reduce the disk load of a new primary.

- *Cached pages:* Our log contains no information about disk reads or cache hits. Thus it is likely that a page that is being read frequently on the primary will not be in the backup's cache. In addition, given the on-disk structure that we use, there are certain structures that are always needed for the file server to go into service. Currently the backup server has to read these pages synchronously with going into service. This would be unnecessary if such pages were logged and pinned in the backup's cache.

With each successive proposal above, the overhead on the backup increases while the latency within which the backup can take over decreases. If we were to try to get the availability latency under one second, we would need to adopt some sort of a cached page approach.

Note that there is one more optimization available, in the case where log entries must pass through a backup server because that is the only path to one of the box replicas. In this case, the backup could take advantage of the log entries in transit to keep its dynamic state up to date. Again, this is an optimization that we have not implemented. You could also imagine keeping the backup's cache warm by having the primary tell the backup what is in its cache and, in the backup's spare cycles, having the backup read the data from the local replica.

File Token Data

Echo provides its clients with a strong guarantee of global data consistency, but also provides for caching of file system data (including directories) in client machines. The details of the algorithms for doing this are not relevant to the present paper. But part of the mechanism is relevant: maintaining the token database. Echo ensures consistency of the client caches by using a token scheme, related to those used by Sprite [9] and OSF's DFS [10]. Each file may have one write token outstanding or many read tokens. If a clerk wishes to cache information from the file, it must hold a read token. If a clerk wants to cache dirty data, then it needs a write token. When a client requests a token from the primary server, the primary server must call back to all holders of conflicting tokens and wait for them to respond before the new token can be granted.

This scheme means that having the correct token database is very important if the server is to maintain its consistency guarantees. During normal running this is easy—the interesting part of the problem is what to do when recovering from a failure. If we simply maintain the token database in volatile memory in the primary file server, then when the primary fails and a backup becomes primary, the new primary would need to recover the token information from the clients (as is done in Sprite). But doing so is potentially very slow—it is a large burst load on the communications sub-system, and it can involve substantial timeouts if some clients that might hold tokens have crashed. Stopping service while the service contacts each client can take so long that meeting a tight availability guarantee can be impossible. For this reason we decided to replicate the token database in volatile memory in the backup server.

An alternative we rejected was to store the token database on disk. This would have permitted us to recover the token database in the same way as the current state of the file system data. However, writing the disk on every token acquisition would have increased system latency a great deal. Because we use caching and write-behind so extensively there are few times when a client process has to wait for the disk, so we did not wish to add latency unnecessarily.[6]

The token database is replicated on all of the servers for a box. Each time a client requests a token, the primary server updates the database in parallel on each of the servers. The client is not told that the token has been acquired until all of the databases have been updated.

The consistency of the various replicas of the token databases is maintained in a similar manner to file data, using epoch counters. Of course, the algorithm for the token database can take advantage of the single consistent image of stable storage provided by the data replication layer, so it is substantially simpler than that layer's epoch algorithm (described earlier in this paper). The token database algorithm uses three counters: each server (primary or not) has a *volatile* counter for the box (kept in its volatile memory), and there are two counters kept in the box's (replicated) stable storage: *serviceT* and *prospectiveT*. A token database in a server is valid if its volatile counter is equal to serviceT. Whenever the primary server believes that a backup server might have been lost, and whenever a new primary is taking control, it executes the following algorithm .

- Increment prospectiveT in the box's stable storage.
- For each accessible server S for this box, if that server's volatile counter is no less than serviceT, then set that server's volatile counter to prospectiveT. In other words, if the token database on S is up to date, then advance its volatile counter; otherwise leave it behind. (It will later be brought up to date asynchronously.)
- Set serviceT to equal prospectiveT in the box's stable storage. This implicitly marks any other token databases for this box as being bad.

During service a token is not granted to a clerk until all of the servers have updated their token database. Thus each server stores a superset of the tokens that the clerks know about. This is okay because clerks are always willing to release tokens that they don't know they hold.

---

[6] Since all of our servers at SRC have backup servers, we took a shortcut and did not implement reconstruction of the token database from the clients at all. This caused problems early in Echo's development when double failures were relatively common. Because losing consistency in Echo is considered to be a disaster that we want applications not to ignore, losing the token database caused many applications to abort. Implementing reconstruction would have allowed us to get through a double failure by simply having the client programs wait long enough for the servers to reboot.

Configuration and location information

As described earlier, the overall Echo system includes name servers that provide the higher level parts of our global name space. When resolving the global name of a file, we necessarily use data from the name servers to locate the file servers that provide access to that file.

Additionally, we store in the name service a description of which boxes each file server serves (so that the file server can know what to do), and which disks contain which boxes (so that the file servers can find them). But in the nature of the name service, its data is only weakly consistent. It is important that any use of data from the name servers can be validated, to prevent incorrect operation. So, we additionally store the set of disk replicas for each box as part of the stable storage of the box (i.e. on the disks of the box).

Note that this organization is completely scalable. It does not rely on broadcast or global name space searching to find information. The scheme is, however, somewhat difficult to manage as there is no automatic way of keeping the name server current (for example when moving a disk from one server to another), nor for maintaining the back links. If we were to redesign this part of the system we would push for greater maintainability by reducing some of the information kept about disks and box replicas. For example, to find a disk all that is really needed is to know which site it is in. Within a site you can search for the disk by brute force or perhaps by accessing a dynamically created database of disk locations.

6. TIMEOUTS AND LEASES

All reliable systems use timeouts. It is the only way for one part of the system to determine reliably that another part has failed. Explicit failure indications are also useful, to provide earlier failure detection. But since the failure notification itself can fail, only the use of timeouts provides firm guarantees.

One specific pattern of use of timeouts, a *lease*, occurs frequently in constructing distributed highly available systems.[7] A lease is a mechanism whereby one party (the “issuer”) gives another party (the “receiver”) the right to use some resource for a certain interval of real time. During that interval, the receiver can use the resource without reference to the issuer; no further communication or negotiation is needed until the lease expires. If the lease is not renewed, both parties know that it has expired, even if they can no longer communicate. All that is required for this to work is that the clock drift between the parties has a known (reasonably small) bound. If the receiver wishes to renew the lease, it must send the renewal request in sufficient time (including allowance for retransmissions) so that the renewal will be received before the previously agreed expiration time. Because lease renewal is time critical, both the holder and the

[7]The term “lease” was coined by Cary Gray [1], though the abstraction was developed independently by us during the Echo project.

implementor need to be careful that the action of renewing a lease doesn't block waiting for other resources. This is discussed later, when we discuss load control issues.

Decreasing the timeout on a lease increases the likelihood that it will expire because of transient communication errors even though the holder is up and still wants to hold the lease. However, shorter timeouts allow the system to respond more quickly to a failure. The cost of guessing incorrectly that a component is down varies widely depending on what resource is being protected by the lease. In the various situations in Echo, incorrect failure detection can cause:

- A client-visible failure (in particular, the return of an error code to a client program or the loss of client visible data). In these cases we use long lease periods.

- A window of vulnerability, during which the crash of a single component can cause loss of service. If the probability of a component failure during this window is small enough and the rate at which such windows are opened is small enough, this can be acceptable. On the other hand, recovering from some incorrectly detected failures can lead to a very long window of vulnerability; in such cases long lease periods are required.

- A performance glitch caused by the recovery logic invoked in response to the incorrectly detected failure. If the length of the performance glitch is within the allowable limits of latency and occurs infrequently enough, this can be acceptable.

In the rest of this section we examine various places where leases and timeouts are used in Echo and our experience with them.

Clerks with File Servers

Clerks, the service agents for client machines, hold various resources implemented by the servers in order to do write-behind and caching while maintaining file consistency. These resources include cache consistency tokens, advisory lock tokens, open file tokens, and free space reservations. Rather than using a separate lease for each resource, we combined all of the resources owned by one clerk on the service for a box into a single resource called a *session*.

Sessions are identified by a unique ID which is recognized by the primary server for that box and its backups. Note that the implementor of the resources is the service, not the primary server, so each server has to know about the state of the resources associated with the session.

The cost of losing a lease protecting a session is quite high as it means that the client's cache is invalidated. Invalidating clean data is relatively benign because it can be revalidated easily without affecting the client programs (except in terms of their performance). But if tokens protecting dirty data or advisory locks are lost, we might be in trouble. If the relevant pages have not been touched by other clerks in the interim (or conflicting locks have not been acquired), the clerk could recover transparently with a new lease; in other cases, there is no practical recovery. In practice, we didn't write the code to recover from either case of this problem.

If the lease for a session is lost and the clerk cannot (or does not) recover, the error must be reported to the affected application programs, since their data updates have been lost or their locks broken. In the worst cases, there is no simple way to report the failure (since the application has proceeded on that basis of correct completion), so we just invalidate affected file descriptors. Sometimes this is the appropriate behavior (the program terminates, reporting the failure politely, or it invokes its own recovery mechanisms). But at other times the application really didn't need Echo's strong consistency guarantees and would have been happier with the weaker semantics provided by NFS; in these cases Echo's reporting of such lease failures is upsetting. Unfortunately, the clerk has no way to distinguish these two situations in the application.

In our current system we use a 60-second lease period. We could recover more quickly from failed clerks by making this period shorter, but if we did so, the problems mentioned above would reassert themselves. To allow us to lower the lease timeout to 1 second we would have to implement lease renewal on top of a level in the network which allows direct control over the allocation of resources. Our current RPC system, which Echo uses for all communication, would need to be avoided or augmented with new capabilities. For example, a lease period of 1 second, with renewals after 0.5 seconds and retransmissions at 0.1-second intervals would probably work well. For 100 client machines this would mean a load of 200 renewals a second per server, a rate that can easily be handled by modern servers.

Note that even these timeout values will work only on a network with low latency and high availability. If the data file is to be accessible from local AN1 hosts and from a WAN, then the lease timeout would have to vary on a clerk by clerk basis, with the clerks far away having larger time outs. This implies, however, that in order to reduce the availability latency for a client, you must ensure that all clients allowed to get tokens that can conflict with that client must run on machines that are connected via the LAN so that the lease period for the associated clerk can be short.

Primary File Servers and their Backup

The primary file server for a box uses timeouts to make decisions about the state of the backup servers for that box. It does this for two reasons: to determine which servers have current replicas of the token database, and to determine which servers might be tracking updates to the log. If one of these timeouts expires, the primary initiates the algorithms described earlier to move to a new epoch, thus invalidating the state of the timed-out backup server if appropriate.

At the same time the backup servers time out the primary. If they lose contact with it, they attempt to become primary, using the algorithms described above. In our design for fast fail-over, the primary designates one backup as the 'first' backup, and it uses the shortest timeout. This is to avoid having multiple backup servers fight for the disks. The first backup has an open shot at the disks for a limited time.

Incorrect failures of these timeouts is largely benign, in the sense that the worst effects are poor performance and more vulnerability to failures that will reduce availability. So the timeouts here can be relatively short. Nevertheless, care is required in the load control algorithms so that such spurious failures do not become common.

File Servers with Disk Servers

When a file server opens a remote disk, it obtains a lease from the disk server. During the period of the lease, the lease holder knows that no other file server may use the given disk. As long as a file server holds leases from a majority of the disk servers storing a given box, then that file server is the primary for that box.

In order to speed up recovery we refined this simple design. The goal is to overlap any needed recovery, for example reading the log, with waiting for the old owner's lease to expire. To do this a file server can request the disk server to steal disks from an existing lease holder. The new holder, in addition to getting access to the disk, also is told the length of time that is remaining in the old owner's lease. The new owner is not permitted to go into service during this time, but it can usefully replay the log and validate its token database. This allows for faster recovery.

In the current system the timeouts for remote disks are 3 seconds. This turns out to be at the limit of what can be implemented using our RPC system, even though the number of disks per server is quite small. We experience a fair number of spurious lost leases on remote disks.

Disk Servers with controllers and disks

The MSCP/SDI I/O system implements two kinds of timeouts. An MSCP controller can have a host timeout. If the controller is not accessed within the timeout, the controller can go offline. On the controller that we use, this means that the controller resets itself and goes comatose for several minutes. The host timeout for a controller can be as small as one second. We set this timeout to ten seconds.

Additionally, SDI disks can also timeout their controllers. If the SDI disk does not hear from its online controller every twenty seconds or so, the disk turns the port offline. This then allows a dual-ported disk to be accessed from the other port.

Using just this mechanism on dual-ported disks implies that after a failure of a server we will have to wait up to thirty seconds before the other server will be able to take over. We alleviate this problem in some circumstances by arranging to have the controller keep-alive done by the lowest part of the server's kernel, which we call the *nub*. In this way even if the file server process or the operating system crashes, the nub still keeps in touch with the controller. To retain fault tolerance, the nub times out the file server process after one-half second of inactivity. If the timeout expires, the nub instructs the controller to release the specific units being used by the server. In this way as long as the nub keeps working, the timeout is reduced to half a second. Initially this optimization was very important because most of our crashes were in the file server itself. However in the mature system most crashes are caused by hardware bugs, which crash the nub as well as the server process.

Note that the disks themselves do not implement leases—they are allowed to go offline at any time, and can then be brought online by the other machine. Initially we dealt with this problem by having the disk server wait for the lease period before returning a handle on a newly online disk. Now it is handled by having the server return the lease time as the length of time remaining in the old owner's lease.

Some of this detail may be viewed as archaic, since modern disk interconnects such as SCSI are quite different. But in reality, equivalent problems (with slightly different parameters) occur in such systems too.

7. RESYNCHRONIZATION

Resynchronization is required when a component of the system is entering service and needs to obtain state from the other components of the system. In particular, this occurs:

- when a component has failed and its replacement takes over;
- when a component becomes accessible again after being rebooted, or restarted, or after the repair of a network partition;
- when a new component is configured into the system.

Sometimes the desired state can be acquired completely (in which case a failure that caused the original component to die can be masked) while at other times the available state might be incomplete (in which case a failure might have to be reported to the client application programs). This section of the paper provides some details on what happens when each of the Echo components resynchronizes.

Resynchronizing a Backup Server

This is the situation where an Echo file server is initializing, and is going to be a backup server for a box that has a working primary server. It can occur when a file server is rebooted, or when it is reconnected after a network partition, or when a new file server is being configured. When a server starts it reads its configuration database and determines which boxes it is supposed to serve. It then attempts to gain ownership of all of the replicas of those boxes. If the box already has a primary it will not be able to gain ownership of the disks, so it enters a receptive state, where it is amenable to being recruited as a backup server for the boxes it is configured to serve.

Meanwhile the primary for the boxes will have been trying to contact the backup server. Eventually it will succeed, and will proceed to help the backup to re-synchronize, in other words to acquire enough state to be able to provide service if called upon.

Once contact is achieved, the primary will start to update the backup server's copy of the token database. Unfortunately the token database is too large to ship over while the database is locked; that is, it takes so long to ship it that locking it for the duration will affect availability as other clients will not be able to acquire new tokens. Instead, the primary locks one file at a time and ships the tokens held on that file to the backup. While this file is locked, further token acquisitions on it are blocked. While this technique is easy to understand and program, on heavily loaded servers, shipping the token database can take tens of minutes. A better approach would be to ship over many tokens with each call. An alternative to fine grain locking is to use optimistic concurrency control, forcing a retransmission if there are requests for the data. We do not know if this type of locking would improve performance significantly.

The data culled from a box's log are not shipped over to new backup servers. New backup servers just start reading the log at whatever position the primary server is currently producing it. The backup need only wait for all of the existing dirty pages to be cleaned for it to have enough information to avoid reading the log if the backup were to take over as primary. The backup can recognize when this occurs: it has enough information when the current log start is beyond the point where the backup commenced processing the log. This strategy is a good one if the log flushing does indeed reach this point before the next failure.

Resynchronizing a primary server

After the primary server for a box fails, and assuming there is an available backup server, one of the backup servers should become the new primary server. This involves several steps.

First, the backup server must obtain leases from a majority of the box replicas.

Next, the new primary has to find the start and end of the log. This tells it whether its dynamic data structures are valid (i.e. they reflect the entire contents of the log). If they don't, the server must read the log from the beginning. In our configurations this takes between ten and twenty seconds on an active box. If the dynamic data structures are valid then the new primary need only read the end of the log, that portion not shipped to it from the old primary. Since the new primary already knows the end position of the portion of the log it has applied, it can seek directly to the correct position in the log and read sequentially.

Having validated its dynamic data structures, the new primary can examine the higher level data structures stored on the box. For example, it can read in the allocation bit map and the File ID map, a map from file ID's to disk addresses of the page containing information about the file.

The next task is to acquire a valid token database. The new primary reads in the epochs that control the validity of the token database from the box. If the new primary's token database is up to date, that is its volatile epoch is greater than or equal to the service epoch stored on disk, then it knows that its volatile token database stores a superset of the tokens known by the clients to be valid. If the token database of the new primary is not up to date then the server must do one of the following:

- Release the box replicas and let another server that might have a better token database become primary. We do this if the primary knows that there is a server with a better volatile epoch. Doing so can cause live-lock if the other server cannot become primary for some reason, such as a broken path between the server and a disk.

- Copy a newer token database from another server. We haven't implemented this. This can be important if the server with the better token database is, for some reason, incapable of becoming primary for this box. Choosing between this technique and the above requires a heuristic for determining which is more likely to succeed.

- Reconstruct the token database by asking each clerk which tokens it holds. We don't implement this but it is important, especially in a system without backup servers. Reconstruction of the token database from clients is likely to take a long time and will probably cause the system not to meet its real-time availability requirements, but at least no errors need to be given to client programs.

- Cause a nontransparent failure (a failure that might be visible to the client programs), and continue without an up-to-date token database. In the last resort you have to be ready to do this. Since we haven't implemented the previous two techniques in the running system, sometimes we have to do this when it is not strictly necessary.

There are several problems with implementing token recovery from the clerks:

- How do we find the list of clerks that might have tokens? We have considered storing a superset of the clerks on disk. To cut down on I/O traffic the list might include some clerks owning no tokens. As this list changes infrequently, keeping it on disk should not cause a performance problem. A special case arises when a clerk becomes disconnected and loses its lease. In this case we must remove the clerk from the list on disk before granting any conflicting tokens. Otherwise after loss of the token database that clerk could become involved with the reconstruction process and regain tokens that were lost, thereby allowing clients of that clerk to read old data.

- How do we handle faulty clerks that lie, saying they own tokens that they do not own? First of all, to obtain a token a clerk must have a user that is authorized to obtain the token. For this to happen a privileged user must have logged into a faulty machine. In general, when this happens, any file that this user can read or write becomes compromised as the clerk can then issue read or write requests that look as if they are coming from the user. But also, during reconstruction of the token database clients of a good clerk may see a non-transparent failure if they want to access files that are also accessible to the users logged into a bad machine. This could happen if the clerk for a bad machine said it had tokens that conflicted with those owned by the good machine. If the server chooses to believe the bad machine over the good machine—it has no way of knowing which is which—the good machine loses. (Note that it will not see a violation of safety, just an unnecessary denial of service.)

After a failure, each clerk that had operations outstanding at the time of the failure must participate in resynchronizing the primary. When the clerk finds the new primary server, it resubmits the uncommitted operations. This happens after the token database has been validated or rebuilt. Reads are of course no problem and can be resubmitted with no further ado. Updates are a different story.

Clerks maintain a pipeline of updates between themselves and the server. This pipeline allows the network transmission time and disk latency to be overlapped with the processing of the updates. Each operation is numbered and the server promises not to commit operation N before all operations numbered less than N have been committed.

When a new primary takes over, the pipeline has to be restarted and the outstanding updates have to be resubmitted.

An open question in our design is how to handle faulty clerks during the recovery of disk space reservations. Disk space reservations allow a clerk to know that enough disk space is reserved on the server to allow the queued write behind to be performed; they are essential to our write-behind mechanism, since we wish to give the guarantee that a write operation will not fail after it has been accepted by a clerk, unless there is an untolerated failure. These reservations are held under the session lease; but unlike the tokens, they are not replicated on the backup. This is because the disk space reservations change so frequently, with every update operation, that the extra messages to the backup server would be too expensive. Because the reservations are not replicated, they must be reconstructed from the clerks. Unlike token database reconstruction (where a faulty clerk can cause problems only for clients sharing files with users of the faulty machine) in this case the faulty clerk can cause trouble even if it is used only by a user authorized to allocate space.

As above, the effect of a faulty client asking for too large a reservation is “merely” to cause a non-transparent failure to clients holding write behind on the box. That is unfortunate but within the semantics guaranteed by the system.

Resynchronizing a box replica

When a box replica (i.e. a disk or disks containing replicas of the box’s data) recovers one of three conditions might hold:

- Hopefully, the primary is still waiting for the box replica to recover and has not advanced the epochs on the other replicas yet. In this case the replica can immediately be placed back into the pool of active replicas; any in-progress writes are redone. In our current system the primary is willing to wait up to 30 seconds for a disk replica to recover after a failure. During this time no new updates can be done. If we were to try to reduce the availability latency below 30 seconds, we would of course need to reduce this waiting time. Making it large has the advantage that more glitches can be handled without resorting to the schemes described below. Note that in our system the failure of the current owner of a dual-ported disk looks like a very short failure of that disk. Because we have not implemented catch-up recovery, described below, we consider it important to respond to that failure without marking the attached disk as out of date. This means having either very large timeouts or very fast switching of a disk from one owner to another.

- Possibly, the primary has marked the recovering box replica as being bad but the log has not wrapped since the recovering replica was last up to date. Echo's logging techniques ensure that all changed pages appear in the log. By going back in the log to the last point where the replica was up to date, we can play the log forward and get a complete list of the pages that should be updated on the recovering replica. The Echo implementation uses a fixed size 2 MByte log. Given our usage, the log tends to wrap around every day or so. So a failure that is recovered in less than a day can be handled by a log-based recovery. Note that the reason the logs can be so short is that for large writes the data itself is omitted from the log, going directly to the home location; only the page numbers are written to the log. We have not implemented this form of replica recovery, which we call 'catch-up recovery,' though it should be a straightforward subset of the recovery technique given below.

- The worst case occurs if the primary has marked the recovering box replica as being bad and the log has wrapped since the time the replica was last up to date. In this case we must do a page-by-page copy of the data and log from a valid replica of the box to the recovering replica. This copy need not lock out other updates to the box. We call this 'full-copy' recovery. In our current environment with 2.3 gigabyte boxes, a full-copy recovery takes about 100 minutes on a lightly loaded server. The recovery code is tuned not to interfere with the normal operation of the system. This might be the wrong trade-off if you were trying to minimize exposure.

One way to look at full-copy recovery is that it is an additional 100 minutes added to the mean time to recovery (MTTR) for a disk. If disks most often become inaccessible due to software failures, this extra 100 minutes is significant. But if disks generally become unavailable due to failures in the disks themselves, then the MTTR for the disk is probably already measured in hours, if it is not days, and the extra 100 minutes are not significant.

Note that with mass market SCSI disks, repairing replicated disks is often not worth the effort. A better strategy is to replace a failed disk with an on-line standby and recycle the old disk for some other purpose. In this case as well, it is only full-copy recovery that is of interest. In our environment the data center disks we use are too expensive to keep hot spares. In addition, since we have dual ported each disk directly to the file servers, the cabling problems in swapping disks are hard to handle. If we had been using disk servers with mass market disks this would have been a viable alternative. On the other hand, mass market disks, which are generally daisy chained together, have additional failure modes not present in data center disks, which generally plug directly into the disk server.

8. LOAD CONTROL

Why is load control an availability issue? First, if the offered load is too large, we cannot meet every client's latency requirement and some clients will see the service as being unavailable. Second, without load control the server can get so overloaded that it can run out of important resources, which can cause leases to fail and can lead to deadlock. Even in the absence of absolute resource limits, the server might also spend so much time queueing requests that it has too little time getting work done. This would cause it to fail to meet its throughput requirements and therefore become unavailable.

If we consider the throughput requirement, the responsibility for load control falls primarily on the system. Because the system controls the clerks and the clerks control when requests are presented to the server machines, the system can, in principle, control the load so that the server machines do not become overloaded. But note that as the offered load increases, the latency for an individual client operation also increases, because limiting the load on the server requires blocking the client requests inside the clerk.

There are two potential problems with load control, one of which comes up in practice and has a solution, the other of which doesn't come up in practice and doesn't appear to have a general solution.

- The load on a client machine is unbounded. If the clients are loading the system with enough work, a client machine might get so overloaded that its clerk ceases to function efficiently. This could be avoided if we used a real-time kernel and the clients were restricted in their use of high priorities. We have had several instances in our use of Echo where the client ran out of either CPU or network resources and lost contact with the service. We have not really solved this problem, but we have avoided it to a large extent by adding additional memory to the client machines.

- A new clerk just starting up will present some initial load on the server until it can be told to back off. If a large number of new clerks all start up over a short period, the load on the server and network could be so large the the throughput would drop enough to cause the system to be unavailable. This has not occurred with Echo. The overhead of a new clerk contacting a server and being told to back off is low enough so that this is unlikely to be a problem.

Limiting the load on the servers by having a server tell a clerk to try the call again later, has complicated consequences in a system like Echo that allows clerks to hold resources while waiting for other server operations to complete. The simple approach of allowing only a fixed number of requests to be outstanding on a server at a time doesn't work. For example, one clerk may be holding a write token requested by many other clerks, but before it is willing to release that token it must do some number of writes to flush its cache of dirty data protected by that token. Using a simple limit on the total number of outstanding requests leads to deadlock.

The consequence of this observation is that it is insufficient to perform load control only in the communication systems, or indeed at any level below the Echo file server. The load control algorithms need input from the server algorithms, to determine which incoming requests should be rejected because they will lead to additional load, and which must be accepted because they are required for making progress.

We implement our load control scheme by reserving resources for different categories of request and guaranteeing that at least that many requests for each class may be outstanding. Some of the quotas are per server while some are per clerk. To avoid deadlock a clerk must ensure that it has no more than the maximum number of outstanding calls outstanding simultaneously. A given request can fall into more than one category in which case it counts against both quotas. Currently the classes are:

- Bulk data transfers. In Echo, bulk data transfers are done with parallel RPC's. Measurements show that maximum throughput for large transfers comes when using seven threads, if the system is otherwise unloaded. We use heuristics based on system load to determine the number of threads to use for a transfer. The quota on the total number of bulk data transfers outstanding is per server.
- Updates outstanding, per server.
- Reads outstanding, per server.
- Token requests incoming (requests by clerks to server for new tokens) per clerk.
- Token requests outgoing (requests to clerks by server for release of held tokens) per clerk.
- Lease renewals, per clerk.
- Number of clerks, per server.

In order to avoid increasing overall system latency, we tried to configure the system so that load control is needed only to flatten out the highest peaks of load, and so that in general load control would not actually throttle back clerk activity.

We did our Echo beta testing with ten users. With this number of users existing operating system limits on the number of requests plus the limits on the number of threads doing bulk data transfers provided sufficient load control.

When Echo was put into service with sixty users, the need for more sophisticated load control became apparent. We had many circumstances where overload caused clerks to become disconnected from the service and lose their tokens, where servers would cease to perform any useful work, and where the added load due to the failure or recovery of a server would cause fatal overload.

With our current load control scheme, described above, we have eliminated all of these problems. However we have not tuned the scheme well enough to allow for maximal performance when the load is moderate; in other words, our load control artificially limits load at levels lower than what could be handled by our existing resources.

9. ONLINE RECONFIGURATION

During the life of the system, there will be many times when administrators need to change it. These changes include software and hardware upgrades as well as changes to management parameters such as the allocation of boxes to disks or of files to boxes. In order to meet our availability goals, we need to be able to make these changes while the system is running.

Fortunately, most such changes are easy. For example, there is no problem with updating configuration information held in the name server while the file servers are running—provided the file servers and clerks are coded to tolerate the transient inconsistencies that occur while the name service is propagating the updates, and to honor the new configuration information when it appears. Equally, it is easy to upgrade a file server's hardware or software provided there is a working backup server for each box. The changes we support comfortably are:

- Adding and deleting a box replica from the set of replicas for a box.
- Reclaiming the disk storage used for storing a box replica.
- Increasing the log area of a box.
- Changing the number of up-to-date replicas needed to allow updates on a box.
- Backing-up a box to offline or online storage.
- Manually changing a primary to a backup server.
- Adding and deleting servers.
- Changing the number of updates that can be committed on one up-to-date replica while not yet being committed on others.

- Changing the name of a box.
- Creating, deleting and renaming volumes on a box.

In addition we had hoped to allow the following change, but never implemented them:

- Moving a volume between boxes while the volume is being accessed. The omission of this feature turned out not to be too much of a problem. Moving a volume without this facility entails turning off write access to the volume while the move is in progress. For users' volumes this meant coordinating the move with the user.
- Splitting a volume into two separate volumes. This facility is used to recover from mistakes in partitioning the file tree into volumes. It is needed either when a volume gets so large that it won't fit in a box, or when the access rate gets so high that a single server cannot serve it adequately. We have had no need for such a facility in Echo, perhaps because of the relatively short period of use and the active attention we gave to dividing data into volumes.
- Cloning volumes (as in the Andrew File System [5]). This would have been most useful as an easy way of creating a snapshot of a volume for backup. Since we did not do volume cloning, did not want to shut down a volume during a backup, and wanted the snapshot semantics, we ended up writing our own backup scheme. This turned out to be a mistake as it diverted us from our main goals and made recovery of user files after a disaster dependent on the correct functioning of our new and seldom tested backup system.
- Growing the data area of a box. This can be done either by adding disk space to existing box replicas or by adding new larger box replicas and the deletion of smaller ones. This would have allowed boxes to grow to accommodate larger volumes or additional disks without the administrative overhead of creating a new box and spreading the load to the new box. In our experience this feature would probably not have been worth the effort of implementing it. Once we figured out the best box size for our system, we configured all boxes correctly. Having this facility would have allowed us to fix up some of the initial boxes we made.

The only significant problems we encountered in this area are not really specific to highly available systems, although they are more apparent in this context. In a system of the complexity of Echo, it is hard for an administrator to predict the consequences of his actions. In particular, there were several occasions when we suffered a failure because the administrator (actually one of the authors) was trying to make a configuration change and inadvertently took down both the primary and the backup server for a box. What is really needed is a higher level program that refuses to allow the operator to make configuration changes that interfere with availability. But pursuit of this topic would be a research project of its own.

10. DETECTING, REPORTING AND REPAIRING FAULTS

When the owners and administrators of a system configure it, they make various decisions about the extent to which they will replicate disks, servers and network connections. They make these decisions so as to provide the level of availability for which they are willing to pay, by tolerating an appropriate number of faults.

A corollary of this behavior is that when a fault occurs, it must be repaired promptly—until it is repaired, the system is less tolerant of further faults, and might not meet its availability goals. So the designer must include mechanisms that will lead to prompt repair of faults. There are three parts to this process: detecting faults, reporting them, and repairing them.

Detecting faults

It's generally quite easy to detect a fault in a component that is currently in use: the detection takes the form of an error return while accessing a component or via a time out on a response to a message. This type of fault results in the usual fail-over behavior as has been discussed above. The only potential complexity here comes from the fact that some error reports can be quite non-specific (e.g. a communication timeout), so localizing the fault to the actual component that needs to be repaired is occasionally tricky.

The more difficult fault to detect is one in a component that is not currently in use, because it is being kept as a backup for recovery from some other failure. But of course it is important to detect these faults before the backup component is needed, or availability will be compromised. There are two common techniques for detecting such faults: periodically exercise the backup components, and periodically provoke artificial failures in the primary components just to see if the backup takes over successfully. (If it doesn't a fault has been detected, but the original primary can resume service so availability is preserved.) Periodic testing has the advantage that it can often be done without interfering with overall behavior of the primary components of the system. It has the disadvantage that it might not put the backup component under enough load to catch certain kinds of faults. We use these two techniques to detect faults in the Echo components as follows.

- *Disks:* With replicated disks, all updates are always written to both disks, so any error in the disk's write logic will be quickly detected. On the read path, things are more complicated because you might do all reading from one disk. In Echo, we use disk arm optimization and load balancing to spread the read load between the disk replicas. Disk read or write faults are noticed by the primary file server and written to a diary file. If there are repeated faults on a disk, the box replica being stored on that disk is eventually removed from the set of active replicas. There is a daemon that periodically gets the status of each box and checks for any replicas that are doing poorly.

- *Paths to the disks:* In the typical Echo configuration in use at SRC, we dual-port each disk to two servers. Thus the path that is in use to a disk depends on which server is primary. The DEC disks that we use have a facility that allows for some checking of the out-of-use disk path. A special command can be sent to the disk by its controlling host, causing the disk to send its status to the offline host. Many DEC systems send this command to the online disks periodically to keep the offline host up to date on the status of the disk. If a report is not received by the backup after a timeout period, we can assume that either the path to the disk is broken or the primary is down. We have found, however, that this technique does not catch many types of disk faults. We have had a number of failures where a path to the disk will work for short status commands but fail on some portion of the long data transfer commands. To detect faults of this type we need to use artificially provoked fail-overs. Our plan was to have the primary and backup switch roles every day or so. This was never implemented, and currently we do not have any automatic means to detect failure in the disk paths not in use. There have been circumstances where a fault in a disk path has been hidden for several days, although this has never caused a loss of service.

- *Servers:* There are certain server faults that can be detected only when the server is under load (i.e., when it is the primary server for a box). Under normal circumstances each server is being used as the primary for some set of boxes. This property is not maintained automatically, but the operators attempt to maintain it manually. We have a mechanism to automatically maintain the server load balancing, but we never felt comfortable enough with it to turn it on during normal service.

- *Network:* The normal timeout algorithms in the communications system will detect faults in the network currently in use. We can distinguish these from server failures in most situations by verifying connectivity to other hosts, such as the name servers. As mentioned earlier, our network includes redundant paths, including redundant network cables into each host. The lowest level of the networking software periodically verifies the integrity of the alternative paths. We have not had problems with fault detection in this part of the system.

These two techniques (explicit error reports and deliberate testing of fail-over) are valuable, but they are not a complete solution. They do not give an end-to-end check on the correct functioning of the system. To achieve this, we maintain several daemon processes on various machines in our environment. These daemons monitor several aspects of Echo's behavior and generate fault reports if the behavior is unsatisfactory. It is these daemons that we have found most useful in getting timely notice of faults. The daemons we use are as follows.

- *Server status daemon:* Reports if a server is down or not serving any boxes.
- *Box status daemon:* Reports if a box does not have its full complement of up-to-date replicas
- *End to end availability daemon:* Reports if a box is not meeting the client latency criteria. It uses quite a long timeout on its operations, so it is unlikely to report load-related problems. This daemon catches certain errors that would not be caught by the other daemons, such as deadlocks within the system that block the execution of file updates.
- *Backup status daemon:* Reports if the backups are behind schedule. This happens if a tape drive has jammed or if the operator has not refilled the tape stacker.

The daemons can be replicated if appropriate.

Reporting faults

Detecting a fault is only the first part of the problem. We must also ensure that the fault is brought to the notice of someone who will get it fixed. In unreplicated systems this is easy: users notice that they cannot get their work done and call the operator. But on a replicated system, users continue to get their work done and the operator needs a new way of getting information about which services are in need of repair.

Writing a fault report to a log file or diary is ineffective. Administrators and operators rapidly cease to read such files. Such reports are still important, but as a way of getting more details once the operator has been notified, or as a way of keeping records so that patterns of faults can be noticed. Equally, mechanisms such as a red light on a cabinet are ineffective: most of the Echo hardware resides in machine rooms that are usually unattended.

We have found that the only effective mechanism is to send electronic mail to a list of people whose responsibilities include dealing with the fault. We do this from the various daemon processes running on other machines that watch Echo components. When a component fails, the daemon sends a message. But even this mechanism must be used with care: if too many messages are sent, the operator learns to ignore them. (When Echo first went online the backup system sent so many messages that they were ignored by the operators, and because of a load-related problem that was exacerbated by our online backups, we got nearly two weeks behind on backups before we noticed.) Each of our watching daemons keeps track of the state of a box or server and reports only on changes to that state. This reduces the volume of mail enough so that the probability of an individual message being important is quite high.

Repairing faults

Repairing a failed component can be as simple as plugging in a power cord that had been accidentally pulled out, or it can involve calling field service or reconfiguring the system to isolate the damaged component. The most frequent repairs we do are as follows:

- Reboot the machine because it has had a parity error or software error. Our system is not set up to auto boot because the home-grown operating system we use has no logging mechanism to keep track of which error occurred.
- Reseat or replace the machine's processor or memory boards to reduce the opportunity for flipped bits or to replace bad chips on the memory boards.
- Reconfigure a box to remove replicas. This is done to increase availability when it is determined that the replica won't be coming back anytime soon. This could have been done automatically.
- Switch a disk from one controller port to another to ascertain whether the disk or the port is faulty.
- Swap the cables going from host to disk unit. After the swap we fail over to the other machine to see if the disk returns.
- Replace various disk components. This takes a while, since it involves a call to field service. The replacement of some components does not lose the disk contents, and so once repaired the disk is automatically reconfigured into the system. However in other cases we have to manually re-initialize the new disk, delete the old replica, and create and add in a new replica into the box.

11. WRITING A HIGHLY AVAILABLE APPLICATION USING ECHO

Echo was designed to make the file system highly available; it does not directly aid in making an application highly available. However, an application can use two aspects of Echo's highly available file system to achieve its own availability goals more easily:

- Echo provides highly available stable storage. As long as the availability and performance requirements of the application are met by Echo, this obviates any need for the application to replicate its permanent state.
- Echo provides highly available locks. A replicated application can use these to coordinate the behavior of its replicas.

SRC took advantage of these features in two separate projects: the Vesta source code control system and the Ivy text editor.

Vesta stores all of its persistent data in Echo. Vesta runs as a daemon process on each machine that has Vesta clients. The different instances of the Vesta daemon synchronize using Echo advisory locks. Vesta uses an Echo lock to protect a log file containing information about the creation of new packages and package versions, and another Echo lock (per package) to synchronize the storing of new versions of a package in the file system. Each time the Vesta daemon claims a lock it examines the state of the file system, and if it finds some work left undone by the previous holder of the lock, it backs out or completes that work. It then proceeds to do its own work. In this way Vesta builds distributed atomic operations without itself implementing a transaction system, distributed locking, or stable storage.

The Ivy text editor maintains for each user a journal of changes made while editing, to support undo operations and to implement automatic replay of edits when recovering from a crash. Ivy uses Echo to store its journal and uses Echo's locks to make sure that only one instance of Ivy is using the same journal. Like Vesta, when Ivy starts up and acquires the lock, it reads the journal and backs out or moves forward any updates that were in progress when the user's last instance of Ivy died.

Another possibility, which we have not explored, is to build a replicated service on top of Echo. A single primary server could provide the service and some number of backup servers wait in case the primary fails. In this situation the primary server would hold an Echo lock, and the backups, running on other machines, would simply wait for the lock. The facilities provided by Echo make it unnecessary for higher level services to implement their own lease and election mechanisms. The fact that Echo maintains both the lock and the persistent data used by the server is critical in making highly available applications easy to program.

12. RESULTS AND CONCLUSIONS

The preliminary design of Echo was complete by late 1988 and coding started in early 1989. Echo was in use by its three implementors in June 1990 and by 15 hardy users in early November of that year. By January 1991 we had turned off our old file system and Echo was the main file system for 60 researchers. Echo was decommissioned in late 1992.

The source code for the Echo file system consists of 125532 lines: 17401 of them purely for the clerk, 100417 purely for the server. This includes all the code for the filestore from the operating system's file system switch down to the low level disk driver, each of which were written afresh for Echo. It does not include the backup system, the name service, management and fault detection programs, mechanically generated code, and pre-existing networking, security, concurrency, and data structure libraries.

To provide file service to our 60 users we had 12 boxes, averaging 2 GBytes of data per box. Each box had 2 replicas, and each was served by one primary server and one backup server. Each disk was dual-ported. There was a total of 48 GBytes of disk storage and 8 file servers. Our file servers and workstations were Fireflies (experimental multi-processor VAXes with an average of 4 processors, each processor being roughly 3

MIPS). The operating system in the workstations and on the servers was Taos (an experimental system that includes kernel threads and binary compatibility with Unix™).

In practice, Echo did not succeed in providing high availability, by any absolute measure. Our previous file servers, which did not replicate users' private data and used *ad hoc* replication for shared data, were widely viewed by our users as having been more available. There seem to be several causes for this:

- Our communication code (RPC and lower levels) was not engineered well enough to withstand the load that Echo placed upon it. As described earlier, this led to a variety of detailed problems, resulting in lowered availability. Most of the causes of this are readily fixable, but there are nevertheless some important lessons here. One is that there must be an interaction between the communication layers and the higher layers to decide how to respond to overload—merely discarding load is liable to have the wrong effects in some cases. Another lesson, often learned but as often forgotten, is that making the code handle the normal case correctly is not enough—the code must behave well under extreme stress, because that is what will be happening when availability is at risk.

- We implemented the Echo code making generous, even casual, use of powerful tools such as garbage-collected texts and an RPC system that would pass complex data structures. The convenience of such tools led us to fail to consider adequately their costs—in performance, overall system complexity, and failure modes that are hard to understand.

- We implemented our most complex system at a time when our hardware was past its prime. Many failures were caused by such trivia as undetected memory errors or melted cables.

- Our old file servers were much more lightly loaded than the Echo servers. This was because the old servers were used only for shared data—private files were kept on local disks or on shared timesharing systems. Simultaneously with converting to Echo, we converted to keeping all of our users' files on the file servers, but we were not adequately prepared for the consequently much greater load.

Those causes are all in some sense avoidable. But some of the lessons we learned are more fundamental, and will require careful consideration by future designers.

- There was an unforeseen multiplicative factor in our failures. Since applications access files from numerous parts of our global namespace, failure of a box containing any of those files will cause the application to fail. So to be merely as good as a previous single-server file system, the availability of each box must be much better than the availability of the previous file servers.

- There was an unforeseen interaction between strong data consistency and availability. If our data semantics had been as weak as those of NFS, applications would have been able to tolerate the failures we did encounter with less disturbance to the users. But since our semantics included a guarantee of data consistency, we were obliged to report as errors events that other systems blithely ignore.

- Sooner or later, your system will have to tackle the problems of load control. Retrofitting load control features is difficult, since they are pervasive through many levels of the software. So the designers' lives will be easier if they consider load control problems throughout the design of the system.

- A system like Echo becomes surprisingly complex. Each idea and each component is simple and understandable; but the aggregate produces complexity that is close to being too much for implementors to handle.

Overall, we believe that the Echo experience has been instructive and valuable. The techniques and insights described in this paper should make it feasible for future file system designers to produce a system that simultaneously achieves high availability, strong data consistency, global naming and good performance.

REFERENCES

1. Birrell, A., et al. The Echo Distributed File System. *Digital SRC Research Report 111.*
2. Gray, C.G. and Cheriton, D.R. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. *Proc. 12th Symp. on Operating System Principles,* ACM SigOps, (Dec. 1989), 202-210.
3. Hisgen, A., et al. Granularity and Semantic Level of Replication in the Echo Distributed File System. *Proc. Workshop on the Management of Replicated Data,* IEEE, (Nov. 1990), 2-4.
4. Hisgen, A., et al. New-value Logging in the Echo Replicated File System. *Digital SRC Research Report 104.*
5. Howard, J. et al. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems, 6,* 1 (Feb. 1988), 51-81.
6. Kistler, J. and Satyanarayanan, M., Disconnected Operation in the Coda File System, *Proc. 13th Symp. on Operating System Principles,* ACM SigOps, (Oct. 1991), 213-225.
7. Lamport, L. Private Communication
8. Mann, T. et al. A Coherent Distributed File Cache with Directory Write-behind. *Digital SRC Research Report 103.*
9. Nelson, M. et al. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems 6,* 1 (Feb. 1988), 134-154.
10. OSF, *OSF DCE 1.0 Application Development Reference, 2,* (Dec. 1991).
11. Popek, G., et al. LOCUS: A Network Transparent, High Reliability Distributed System. *Proc. 8th Symp. on Operating System Principles,* ACM SigOps, (Dec. 1981), 169-177.
12. Schroeder, M., et al. Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links. *IEEE Journal on Selected Areas in Communications, 9,* 8 (Oct. 1991), 1318-1335.