Proceedings of FREENIX Track:
2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

PERMANENT WEB PUBLISHING

David S. H. Rosenthal and Vicky Reich

Permanent Web Publishing

David S. H. Rosenthal
Sun Microsystems Laboratories
Vicky Reich
Stanford University Libraries

"Diffused knowledge immortalizes itself"
Vindiciae Gallicae Sir James Mackintosh (1765-1832)

Abstract

LOCKSS (Lots Of Copies Keep Stuff Safe) is a prototype of a system to preserve access to scientific journals published on the Web. It is a majority-voting fault-tolerant system that, unlike normal systems, has far more replicas than would be required just to survive the anticipated failures. We are exploring techniques that exploit the surplus of replicas to permit a much looser form of coordination between them than conventional fault-tolerant technology would require.

1. Introduction

In a classic paper Leslie Lamport[lamport] set out the basis for building a distributed system with n replicas which could tolerate f faults, where $f = \lfloor (n-1)/3 \rfloor$, by having the replicas vote in "elections" to decide the system's behavior.

We describe work in progress to develop a solution to an important real-world problem, the preservation for future generations of scientific, technical and medical (STM) journals published on the web. Any system guaranteeing long-term preservation of information must tolerate faults such as disk crashes, network outages and malicious attacks. Certain special characteristics of this problem mean that in our case n is much greater than $3f+1$. The large number of replicas allows us to try a somewhat different approach to fault-tolerance. If the classic approach is analogous to elections, our approach is analogous to opinion polls.

We start by outlining the problem and generating requirements for a solution. We show how these requirements led to a design for which $n >> 3f+1$. We describe the design in some detail and provide a brief report on the status of the implementation. We compare our approach with related work and assess its strengths and weaknesses. We conclude by speculating about the usefulness of similar techniques to other applications for which large numbers of replicas are appropriate.

2. The Problem

In most respects the Web is a far more effective medium for scientific, technical and medical (STM) communication than paper. Stanford Library's Highwire Press[highwire] led the transition of STM publishing from paper to the Web and now publishes the on-line editions of about 180 of the top STM journals. They pioneered techniques such as datasets in spreadsheets behind graphs, dynamic lists of citing papers, e-mail notification of citing papers and so on. These, added to the basic hyperlinks and searchability, make the Web versions both easier to access and more useful when accessed than the paper ones. Web versions frequently appear earlier and contain much more information. Many journals now publish some papers only on the Web.

Librarians have a well-founded confidence in their ability to provide their readers with access to material published on paper, even if it is centuries old. Preservation is a by-product of the need to scatter copies around to provide access. Librarians have an equally well-founded skepticism about their ability to do the same for material published in electronic form. Preservation is totally at the whim of the publisher.

A subscription to a paper journal provides the library with an archival copy of the content. Subscribing to a Web journal rents access to the publisher's copy. The publisher may promise "perpetual access", but there is no business model to support the promise. Recent events have demonstrated that major journals may vanish from the Web at a few months notice.

This poses a problem for librarians, who subscribe to these journals in order to provide both current and future readers with access to the material. Current readers need the Web editions. Future readers need paper; there is no other way to be sure the material will survive.

The transition to the Web will not be complete until librarians are willing to buy Web-only subscriptions. To do so they need confidence in their ability to provide their readers with long-term access to the content they are buying. The problem is in three parts:

- The bits themselves must be preserved. All digital storage media have a limited lifetime; the bits need to migrate from one medium to another over time. In practice it is difficult to fund a bulk copying effort when a medium starts decaying, so all but the most valuable bits are lost at each transition[bit-rot].

- Access to the bits must be preserved. Suppose a reader clicks on a link 20 years from now that doesn't resolve because the only copy is now on a CD in a secure store. Whom does the reader call to get the CD from the store into a specially preserved CD drive? Unless links to pages continue to resolve, the material will effectively be lost because no-one will have the knowledge or patience to retrieve it.

- The ability to parse the bits, once accessed, into human-readable form must be preserved[format].

It is important to observe that there can be no single solution to this problem. A single solution of itself would be perceived as vulnerable. By proposing one solution we are not arguing that other solutions should not be developed and deployed. Diversity is essential to successful preservation.

3. Requirements

Librarians' technique for preserving access to material published on paper has been honed over the years since 415AD, when much of the world's literature was lost in the destruction of the Library of Alexandria[alexandria]. Their method may be summarized as:

> Acquire lots of copies. Scatter them around the world so that it is easy to find some of them and hard to find all of them. Lend or copy your copies when other librarians need them.

In this context, note the distinction between archives, which we are not discussing, and general circulating collections, which we are:

- The goal of an archive is preservation, typically of material that is unique and/or impossible to replicate widely. Access is restricted via locked stacks, access logs and so on to ensure preservation.

- The goal of a circulating collection is access, typically to replicas of material that is widely copied. Risks are taken with preservation to achieve access. Copies are on open shelves and are loaned to readers on a promise that they will eventually be returned.

Libraries' circulating collections form a model fault-tolerant distributed system. It is highly replicated, and exploits this to deliver a service that is far more reliable than any individual component. There is no single point of failure, no central control to be subverted. There is a low degree of policy coherence between the replicas, and thus low systemic risk. The desired behavior of the system as a whole emerges as the participants take actions in their own local interests and cooperate in ad-hoc, informal ways with other participants.

If librarians are to have confidence in an electronic system, it will help if the system works in a familiar way. The fundamental requirement for LOCKSS (Lots Of Copies, Keep Stuff Safe)[lockss] was, therefore, to model their techniques as closely as possible for material published on the Web.

If libraries can take physical custody of the journals they purchase, in a form that preserves access for their readers, they can assume the responsibility for their future. If a library takes custody of a copy of the Web journal, the copy can behave as a Web cache and provide access whether or not it is available from the original publisher. If many libraries do so, the caches can communicate with each other to increase the reliability and availability of the service, as inter-library loan increases the reliability and availability of access to information on paper.

Another way of looking at a system of spreading copies of Web journals around the world is that the librarians running the system are buying insurance for their journal subscriptions. What are the librarians insuring against? Reasons why their readers would lose access to a journal include:

- Failure to renew a subscription, for example because of budget cuts or a price increase by the publisher.

- Change of policy by the publisher, for example because a not-for-profit journal was taken over by a for-profit publisher.

- A publisher going out of business.

- Incompetent or careless management of the publisher's web service.

In all these cases the symptoms are either a refusal by DNS to resolve the name in the link URL or a refusal by the server named to supply the content.

Libraries have to trade off the cost of preserving access to old material against the cost of acquiring new material. They tend to favor acquiring new material. To be effective, subscription insurance must cost much less than the subscription itself.

The biggest journal Highwire publishes generates about 6GB/year. A cheap PC to hold 5 years' worth might cost $600 today, which is about 10% of the subscription for the 5 years. If the running costs of the system can be kept low enough, it should now be practical for many libraries to maintain their own copies. The prospects for this insurance improve as equipment prices fall and subscription prices rise[disk-cost].

In this context, Open Source development is crucial:

- The goal of the project is to inspire confidence. It is hard to have well-founded confidence in a system whose operations are kept secret.

- The system's economics mandate free distribution of the software; there's barely a budget for the hardware.

- The longevity of the system will require many generations of programmers to refine it as problems are encountered.

4. Implications

Conventional replicated fault-tolerant systems are designed around the question "how few replicas are needed to survive the anticipated failures?" This required number of replicas, perhaps 5, are then organized into a tightly administered system.

To solve our problem, each library wanting to insure their subscription must take custody of a copy of the Web journal in question. If the system is successful there will be many more replicas than needed to survive the expected failures. There may be hundreds. It isn't possible to administer hundreds of systems, each under the control of a separate institution, as tightly as the small number of replicas under centralized administration in a conventional fault tolerant system. Nor is it efficient to have hundreds of systems participate in every operation of a distributed system in order to survive the failure of a tiny fraction of them.

In the real world there is no authority controlling libraries. Anyone can claim to be a library. Individual librarians assess the credibility of such claims on the basis of experience. They are suspicious of the long-term dependability even of neighboring libraries whose *bona fides* are not in question. Creating tightly controlled long-term cooperative efforts in this environment is not effective. LOCKSS therefore needs to trade off its surplus of replicas for a looser form of cooperation among the replicas.

5. Design

The design goal for LOCKSS is to provide librarians with a cheap and easy way of running Web caches which pre-load the journals they subscribe to, preserve them for posterity by never flushing the cache, and serve their pages to the library's readers if the publisher does not. The design falls into three parts:

- Pre-loading the cache as new issues of the journals are published.

- Ensuring that readers can access the journals from the publisher or from the cache.

- Preserving the contents of the cache.

The design takes advantage of some unusual features of the Web editions of STM journals:

- The peer-review system requires that the articles are immutable once published. The history of publishing on paper has reinforced this feature.

- The web site for a journal has a logical structure, with directories for each volume, each issue within the volume, articles within the issue, and so on.

- New content is published on a fairly regular cycle that may be weekly or monthly, not hourly.

It should be emphasized that we are not designing a general-purpose Web content preservation system. LOCKSS is designed to preserve only journals published by Highwire; we are in control of both ends of the process and could if necessary alter the HTML Highwire generates to make preservation easier. It may be possible to apply the system to other types of content, but that is not at present a design goal. LOCKSS is clearly not suitable for volatile content.

It should also be emphasized that we are describing work in progress. As this paper is being written we are preparing the prototype for the system's first major test, using a small group of libraries and a single journal. We expect the design to evolve as we gain experience.

5.1. Collecting

A librarian instructs an instance of LOCKSS to preserve a volume of a journal by providing the publisher's root URL for the volume and a frequency of publication, say monthly. At that frequency a web-crawler starts from the root URL and fetches all new pages within that sub-tree. The publisher's web server sees this access as coming from an authorized IP address, so it is allowed. Note that we don't depend on readers accessing the material to populate the cache.

This is off-the-shelf technology; we currently use the *w3mir* crawler[w3mir], which is written in Perl and easy to adapt to our needs. The only change needed was an interlock with the cache preservation code, to prevent content being checked while it was being collected.

5.2. Serving

This is also off-the-shelf technology. The prototype uses the Apache[apache] web server to export the contents of each cache to the local network's users.

Tighter integration with the cache management code will be needed in the future to support the code that prevents the publisher's access control system being subverted [see 6.3.2]. At that point we expect to switch to a simple HTTP server in Java, which can check the cache manager's internal data structures to determine if accesses are appropriate.

5.3. Preserving

The heart of LOCKSS is the process by which caches cooperate to detect and repair damage. The caches communicate using an IP multicast protocol to discover which URLs should exist and what their contents should be. This protocol runs continually but very slowly between all the caches. If a cache discovers a missing or damaged URL it can fetch a new copy via HTTP from the original publisher, or from one of the other caches. Care has to be taken not to subvert the publisher's access control mechanism; content should only be delivered to sites that have rights to it.

The process works this way. A cache will notice that a part of the material it is preserving has gone long enough without being checked. It will multicast a call for a *poll* to decide the value of the message digest of the sub-tree below the directory representing that part. Other caches hearing the call will compute their digests and reply. The caches hearing the replies will tally the poll. If they are on the winning side their cache is intact. If they are on the losing side, their cache contains some damage. The damage is located by:

- Calling a poll to determine the set of names in the directory representing the part containing damage.
- Calling a poll on the message digest of the sub-tree below each name.

The process descends the tree until it finds files instead of directories. If the digests agree the file is intact. If not the file is damaged and a new copy is fetched[1]. The enumeration of the names in the directories also finds any extra names, which are removed, and any missing names, which cause the corresponding file or sub-tree to be fetched.

6. Protocol

We describe the protocol from the bottom up, starting with the basic polling mechanism, then the different types of polls, and then the policies that string the polls together.

6.1. Elections & Polls

We call the inter-cache protocol LCAP [Library Cache Auditing Protocol]. The design was inspired by Scalable Reliable Multicast (SRM)[srm], a peer-to-peer reliable multicast protocol whose importance for LCAP is its use of random timeouts to run a "sloppy" form of election.

All participants in SRM subscribe to and multicast packets to a single IP group address. If a missing packet is detected, a request is multicast for the packet to be re-transmitted. If a request is received for re-transmission of a packet that has been received a random timer is started. If a re-transmission of that packet is received while the timer is running, the timer is cancelled. Otherwise when the timer expires the packet is re-transmitted to the group address.

[1] In production, we'll be more careful. Files won't be overwritten or removed but moved to a backup tree.

This random timeout mechanism is used to elect a participant to perform the re-transmission. It is not a perfect mechanism; sometimes failing to elect anyone and sometimes electing more than one. If no one is elected, the requester times out and tries again. If more than one is elected, the repair is re-transmitted more than once, wasting bandwidth but doing no other harm.

In practice the "sloppy election" mechanism is very effective in those cases where delays several times the length of a packet round-trip are tolerable. It is statistically very likely to elect a single participant. It has built-in load balancing, electing a participant at random. It tolerates faults; if the participant with the shortest random delay fails before transmitting its repair another will take its place. For our purposes these multicast, random timeout based "sloppy elections" have many attractive features:

- Each election selects its own electorate; there is no configuration database to maintain in a consistent state.

- They load-balance automatically. If there are more than enough potential voters the actual voters will be chosen at random.

- The elections survive lost packets very well.

In the real world elections are tightly controlled and very expensive procedures. A register of electors must be maintained, with rigorous procedures for qualifying those who may vote. Voters have to identify themselves at the polls. Long experience has led to many precautions against fraud.

Opinion polls have none of this overhead yet almost always predict the result of an election correctly. They do require careful attention to sampling and question design, but because they don't need the administrative structures nor the mass participation of a real election they are much cheaper and quicker.

LOCKSS uses a variant of the SRM "sloppy election" technique to implement something akin to an opinion poll. The caller of a poll announces:

- the *subject* of the poll, the URL to which it applies, and

- a *hurdle* for the poll, the number of agreeing votes needed to make it valid, and

- a *duration* for the poll, setting the time that will elapse before the votes will be tallied, and

- a *challenge* for the poll, a random string that is prepended to the data to be digested.

These values appear in the header of each packet sent as part of a poll.

A participant receiving a call chooses a random delay in the duration when it plans to vote. The voter chooses a random *verifier* string, prepends the challenge and the verifier to the data, and computes a message digest. When the timer expires it multicasts the remaining duration, the challenge, the verifier and the message digest. The challenge and the verifier prevent replays and force each voter to prove that they have the content in question at the time of the poll.

Votes are tallied at each participant receiving them by prepending the challenge and the verifier from the vote to the local copy of the data and computing the message digest. If this digest matches the one in the vote it is an agreeing vote, otherwise it is a disagreeing vote.

A participant receiving more than the hurdle number of agreeing votes before their timer expires can, if the agreeing votes are the majority, decide that there is no need to vote and cancel the timer[2]. In determining that it agrees with the majority it will have checked its local copy. There's no harm in voting unnecessarily, but participants need not tally excess votes.

There's no "electoral register" determining who can and cannot vote. If a cache has a copy of the data in question it can vote, because it is doing the job of preserving access to the data. This models the real world; there is no authority deciding who is qualified to be a library.

6.2. Operations

The cache consistency checking process uses two different types of polls:

- A *compare* poll asks voters to compute the message digest of the challenge, their verifier and the data in the sub-tree below the subject URL.

[2] In practice the timer is not cancelled but suspended, in case of a late rush of disagreeing votes, see 7.2.4.

- An *expand* poll asks voters to compute the message digest of the challenge, their verifier and the set of names in the directory named by the subject URL. Voters in an expand poll also send the set of names in their directory.

A participant tallying a compare poll will either:

- Agree with the majority, in which case nothing need be done. It has been established that the system as a whole is storing at least the hurdle number of good copies.

- Disagree with the majority, in which case part or all of the local copy of the sub-tree named by the subject is bad. The participant chooses a random timeout and, if another participant has not already done so, calls an expand poll on the subject URL.

A participant tallying an expand poll counts the number of votes for each set of names. The set with the most votes, provided it reaches the hurdle, is the winner. The participant then compares the winning set with their set:

- Names in the local set but not in the winning set need to be removed.

- Names in the winning set but not in the local set need to be fetched.

- Names in both sets need to be compared. For each such name the participant chooses a random timeout. When it expires, provided some other participant has not already done so, the participant calls a compare poll on the name.

In this way the checking process walks the directory tree to locate and repair damage.

6.3. Policies

The system depends on a number of parameterized policies. Over time, our experience with the system will determine if the current set of policies and the corresponding parameter values are adequate.

6.3.1. Rate Limits & Poll Durations

Before we started implementing LOCKSS we expected it to run very slowly just because there was no need for speed. Two things changed this:

- Voting in a top-level poll takes a long time simply because top-level polls typically check an entire issue of a journal, which will range between a few hundred megabytes to a few gigabytes. The voter needs to compute the message digest of all this data once for its own vote, and once for each other vote it checks. The data will normally be hashed about the hurdle number of times. The obsolete 100-200MHz PCs we're using can take an hour or two to do this for our test journal.

- Integrity considerations [see 7.2.2] imply running as slowly as possible is important. The system must run fast enough to compare cached data on average several times between losses, but no faster.

While the need to compute message digests makes top-level polls take a long time, polls checking an individual file could happen much more quickly. The caller of a poll decides on a duration by measuring how long it takes to compute its own digest, multiplying by the hurdle number it chooses, and by a safety factor. The system limits bandwidth consumption by imposing a minimum duration for polls.

6.3.2. Access Control

Each time a cache votes on the winning side it is demonstrating that it has a good copy of the sub-tree in question. Caches remember the winning votes they hear in compare polls for some time, longer than the expected time between failures. They will provide repairs only to caches they remember having voted on the winning side in a poll for the corresponding sub-tree.

In this way they avoid subverting access control. The only way to get a copy other than from the publisher is to have shown in the recent past that you used to have a copy. At some point this recursion arrives at a copy that came from the publisher, and thus satisfied at that time the publisher's access control restrictions.

6.3.3. Maintaining Redundancy

If polls of a given sub-tree consistently fail to reach the hurdle number, this signifies that there aren't enough copies being preserved and the content is at risk. At this stage we believe it is appropriate to notify the people running the system, who can arrange for more copies to be created. At a later stage it might be possible to have LOCKSS instances keep a reserve of disk space in which they can store copies of at-risk material while people cope with the situation.

7. Integrity

So far, we have described a system in an ideal world without malicious participants. Alas, even libraries have enemies[enemies]. Governments and corporations have tried to rewrite history. Ideological zealots have tried to suppress research of which they disapprove. We have to assume that bad guys will try to subvert LOCKSS if it gets deployed.

7.1. Conventional Approaches

One approach to preventing the bad guy rewriting history that is often suggested is to have the publishers sign the articles they publish. Readers could then check the signature to determine that the document had not been modified since it was signed. We encourage publishers to sign articles (they don't at present) but even if they did it wouldn't guarantee preservation:

- For the future reader to get an article, access to it has to be preserved. Signing the articles helps to verify a copy as authentic once one is obtained, but doesn't help to get a copy in the first place.

- For the future reader, or cache, to be able to check the signature on an article and verify that their copy is authentic they have to obtain the publisher's certificate, check that a trusted Certificate Authority has signed it, and that it hasn’t been revoked. This reduces the problem of preserving access to information under the librarians' control (the articles) to the problem of preserving access to smaller units of information outside the librarians' control (the certificates). There's no guarantee that the Certificate Authority will survive the many decades of our timescale.

- The validity of the signature depends on the publisher's private key being kept secret, and on the encryption and hash algorithms involved remaining unbroken. Neither is very likely over our timescale.

Although certificates and signatures are not useful over the long term it is certainly possible to design a system for preserving access in which they are used only over limited periods of time.

Suppose an institution is established which decides from time to time which other institutions are eligible and competent to take part in the preservation effort. It would maintain a registry of certificates a participant could use to decrypt and validate communications from other participants. This system need not use multicast communication, it would have a rendezvous point similar to the Service Location Protocol's daemon[slp]. Participants would use this to locate one another. As institutions were subverted or failed to maintain adequate standards of preservation the registry would be updated to delete or revoke their certificates.

Again, the problem of preserving access to information under the librarians' control has been reduced to the problem of preserving smaller units of information outside the librarians' control. Only this time the design has a single point of failure, the registry. Once the bad guy has subverted the registry the entire system is compromised.

The fundamental weakness of conventional approaches is that they divide guys into good and bad by having some external authority place white or black hats on them. A guy with a white hat is free to subvert the system. Just because an institution is a library, it doesn't mean that everyone will, or should trust it. A university library in Greece might, for example, regard a university library in Turkey, as a reliable source of information about chemistry but unreliable on the topics of Aegean geography or Kurdish history. Equally, just because some years ago a library was trustworthy, that doesn't mean they're trustworthy now. The government funding the library may have been replaced with a less scrupulous one.

7.2. Alternative Approach

The approach we're experimenting with in LOCKSS is to divide guys into good and bad by observing and remembering their behavior over a period of time. For our purposes, a good guy is one who:

- maintains a good copy of the journal content,

- votes in polls to prove that the copy is good and to help others prove that their copies are good too,

- remembers that others have voted in the majority for a long time,

- and supplies good copies to others when requested to repair damage.

A bad guy is one who, among many potential crimes:

- votes too early or too often,

- votes on the losing side of too many polls,

- fails to verify their vote on request,
- or supplies bad copies to others.

Note that these are all public actions, observed by others.

7.2.1. Maintaining a Reputation

Its not important for the functioning of LOCKSS that a participant actually be a library, only that the participant exhibit the behavior of a library over a period of time. Because LCAP is a peer-to-peer multicast protocol, the behavior of each participant is visible to the others. They can observe, remember and make their own estimates of the participant's reliability.

This system has many analogies to "reputation mechanisms" in on-line game environments like Ultima Online[ultima]. Just as in our case, there's no way of knowing "who a participant really is". Indeed, part of the attraction of the game is that players can experiment with multiple identities. The avatar of a particular identity carries a reputation based on its recent actions as observed by other avatars. A bad reputation can be cleaned up over time by performing actions others judge as laudable. Note, however, that games maintain a central registry of avatars' reputations. In the LOCKSS model there is no central registry, each "player" maintains its own registry of other players' reputations.

7.2.2. Running Slowly

Recent actions have higher weight in determining credibility than ancient ones, reflecting the fact that bad guys can reform, and that good guys can be subverted. The latter observation leads to the interesting conclusion that the system must be designed to run extremely slowly, both in the sense of wall-clock time, and the number of operations needed to make a significant change to the cached information:

- Running very slowly limits the damage that a guy who turns bad can do while he retains a reputation based on his actions before he was subverted.
- Running very slowly means that in order to do significant damage a guy must persist in taking bad actions over a long period of time. If a bad guy calls your telephone, your goal should be to prolong the conversation. This allows law enforcement to track the offender down. In our case the system requires the bad guy to take actions that are both public and obviously bad over a long period of time, allowing the good guys time for location and dissuasion. No system can be immune from penetration; systems should be designed to slow the bad guy's progress and limit the potential damage.
- It is easy to mount a denial of service attack against any multicast protocol. Forcing the protocol to run very slowly makes these attacks unattractive to the bad guy. Sending lots of bogus packets to the LOCKSS IP multicast group address will cause polls underway to fail to reach the hurdle and prevent new ones being called. But as soon as the flood stops the system reverts to normal operation. Preventing the system achieving its goal of long-term preservation requires the bad guy to flood the group for years on end.

7.2.3. Detecting Bad Guys

If no bad guys are active and no participants are losing data, top-level compare polls will be taking place infrequently, and each will be a landslide. To maintain credibility, participants must vote on the winning side in these polls. Doing so is hard work; it requires the voter to compute message digests of hundreds of megabytes of data several times over. This is a good thing:

- It achieves the goal of making the system run slowly in wall-clock terms.
- It provides the system with some inertia and requires the bad guy to invest a lot of effort before he can make an impact on its behavior.
- It makes sure each participant's reputation information is up-to-date. This allows the credibility of inactive or subverted participants to decay quickly, limiting the amount of damage subversion can do.

If no bad guys are active but occasionally participants lose data, polls will descend the tree on occasion and some will have a few dissenting votes. This is the normal behavior of the system.

The structure of a typical journal web site places many intermediate directories between the root and the actual journal articles. It takes many polls to descend from the root to an actual article the bad guy might target. If polls are observed close to the leaves of the tree in which the result is close, or even where there are substantial numbers of dissenters, one or more bad guys are active. The people running the participants can take measures to stamp them out.

7.2.4. Preventing Fraud

Each participant in a poll verifies a proportion of the votes they tally to help prevent fraud. Very few votes are verified if the poll is a landslide; the proportion rises as the poll becomes an even contest.

The verifier in the vote is actually the digest of a random string that the voter keeps private. The tallier verifies the vote by unicasting a verification request to the sending address in the vote naming the subject and the challenge. The voter replies with the subject, the challenge and a string whose message digest is the verifier in their vote. Voters failing to verify one of their votes after several attempts lose credibility quickly; it is likely that they are being spoofed.

7.2.5. Advantages

This system is strong in some unusual ways:

- There is no central coordination point that can be attacked. Each participant is independent; acting in its own interests, trusting others only as far as necessary and no further than experience shows them to deserve trust. The design goal is that the only way to subvert the system would be to subvert a majority of the participants.

- The system makes as few demands on the infrastructure as possible. It doesn't depend on services such as the Domain Name System, or a Public Key Infrastructure or some mythical Library Certification Organization. All that's needed is for the underlying network to route IP unicast and multicast datagrams.

- It doesn't depend on preserving any meta-information. Provided enough participants preserve the journal articles themselves, a site can corrupt or lose any or all of its information. The more it does, the less its credibility will be among the other participants for a while.

- It doesn't depend on keeping anything secret for any length of time, especially not passwords or encryption keys. Voters need to keep the string that generated their verifier secret for the length of the poll, but this is all.

- It doesn’t depend on encryption or hash algorithms resisting attack, because it doesn't use encryption. It does use a hash algorithm during a poll, but this need resist attack only for the duration of enough polls to build or destroy a reputation.

- By operating slowly even on human timescales the system makes it easier to detect an attacker and limits the damage he can do before being stopped.

8. Implementation

The prototype's implementation of the LCAP protocol is in Java. It makes heavy use of threads to maintain the context for ongoing polls, and to ensure that time-consuming operations like computing the message digest of a few hundred megabytes of journal data don't interfere with other tasks. The source will be released under a Stanford equivalent of the U.C. Berkeley license.

It uses SHA-1[digest] as the message digest function, combined with a filter that parses the HTML of an article to isolate the part that represents the text the authors wrote. This is necessary because successive fetches of a given article from Highwire do not return exactly the same bytes:

- Some journals place advertisements on their pages; the advertising system selects different ads at different times.

- Some features of the article presentation, such as the list of citing articles, change over time.

The current filter is rather crude, more sophisticated versions will be needed before the system goes into production.

9. Production Use

When LOCKSS gets into production, librarians will have to install new instances, manage them through their useful life and replace them when they fail or fill up.

9.1. Installation

For the prototype, we are developing an installation process based on the Linux Router Project's[lrp] distribution. The librarian will download the image of a generic boot floppy disk, boot it, answer a few configuration questions and then choose an option that re-writes the floppy disk into a configured boot floppy for the new system. This will be write-locked and used to boot the system in production.

Each time the system boots it will start with a clean, known-good system image in RAM-disk. It will then download, install and run the LOCKSS code. The only data on the hard disk that survives across reboots will be the cache contents and meta-data.

9.2. Management

One major goal of the initial tests is to provide the information needed to design a management interface for the system. We don't yet understand what librarians will need in order to understand and have confidence in the normal operation of the system, nor to detect and respond to abnormal events.

9.3. Replacement

When a LOCKSS instance fails or fills up it can simply be replaced by a new, empty instance assigned to the same journal. The new instance will detect the missing data and reload it from the publisher or other caches. To avoid wasting time and bandwidth, we expect to provide a "clone" option that would allow the librarian to nominate an existing instance from which the new instance's cache would be copied.

10. Performance

There are three important performance metrics for LOCKSS once it is deployed in production:

- What does it cost a library to run it?
- How often does the system as a whole lose or corrupt journal articles?
- What is the probability that a reader will encounter a missing or corrupt article?

Credible numbers for these metrics will not be available for many years. The best we can do right now is some back-of-the-envelope estimates of the I/O, bandwidth and failure rates. These encouraged us to go ahead with the alpha test, but are too sketchy to publish. We expect to report measurements from the alpha test when we present this paper.

11. Related Work

11.1. Fault Tolerance

The conventional approach to fault tolerance through a limited number of replicas is brilliantly illustrated by Miguel Castro & Barbara Liskov[castro], who built a replicated, fault-tolerant implementation of NFS that benchmarked only 3% slower than the baseline implementation when no failures were encountered and, of course, infinitely faster when they were.

11.2. Internet Archive

LOCKSS is not an archive, and it does not attempt to preserve general Web content. An ambitious attempt to archive the entire Web is underway at the Internet Archive[archive]. They have currently collected almost 15TB of data, which is primarily stored in a tape robot. As an archive, their mission is primarily preservation, which they plan to ensure by careful treatment of stored data and media migration, not replication. They do not attempt to ensure that the original URLs continue to resolve.

11.3. Intermemory

In the opposite direction, a team at NEC's Princeton labs built a replicated, distributed Internet-scale file system[nec]. Machines joining the system volunteer disk space to the file store, which uses hashing techniques to smear stored information across multiple replicas. This preserves access to files via the names assigned to them when they are stored, and to their contents via replication. This Intermemory system shares with LOCKSS the basic approach to preservation through replication and copying among unreliable storage systems, but differs in that it exports a file system interface rather than a Web interface, and that its internal workings are obscure to the uninitiated.

11.4. Digital Library research

The NSF is coordinating a major research initiative into the general problem of constructing a Digital library[dlr]. Projects funded by this DLI2 initiative address a much broader set of issues than LOCKSS, including versioning documents as they change, a vast range of protocols and formats not just HTTP/HTML, and issues around metadata. Because their problem is much harder their technology is not yet deployable.

11.5. Robust URLs

Thomas Phelps and Robert Wilensky[words] at U.C. Berkeley have discovered that a Web document can be found uniquely with very high probability if a surprisingly small number, from 5 to 8, of carefully chosen words from the document are given to a search engine. They propose that links to documents be augmented with these signature words to provide browsers with a viable fallback if the URL fails to resolve. This is an interesting idea, but it assumes that the document is somewhere accessible to the search engine after its original publisher has failed, and that the search engine has permission to read it.

This insight could usefully be combined with ideas from Freenet[freenet], a distributed, search-based information store. Freenet shares with LOCKSS the goal of a system free of the vulnerabilities of central administration and control, but it does not attempt to preserve information whose value is not related to its popularity, and each server appears to trust every other server to supply authentic copies of data being stored.

11.6. Napster

Napster[napster] provides an interesting example of combining many replicas of a single data item, in their case a song in MP3 form, to form a highly available data resource. Of course, the Napster directory service is itself a single point of failure. The Gnutella distributed directory service would have been more relevant to our problem.

12. Assessment

We stated three goals for LOCKSS. How does the design rate against them?

- The bits must be preserved. If enough replicas can be deployed the system should have a very low probability of losing bits accidentally. The system's effectiveness at preventing malicious actions destroying bits is open to debate. It may be necessary to use encryption and to identify and authorize the participants.

- Access to the bits must be preserved. Readers in participating institutions should have a high probability of having their original links resolve to good copies of articles.

- The ability to parse the bits into human-readable form must be preserved. The process of continual gradual replacement of the software, driven by the need to replace the hardware as it breaks or fills up, allows for format conversion as it becomes necessary.

13. Future Work

We're running an initial test of the prototype for a couple of months with about a dozen instances and a single journal starting in April 2000. We plan to assess this test, incorporate the experience and run a second test at a much more realistic scale later in the year. We hope this test will include an attack team trying to subvert the system.

We're also exploring the suitability of LOCKSS for applications other than journals. One obvious example is the government documents that used to be kept on paper in the "depository library" system, but which are now being published on the Web.

Broader applications of the underlying model of fault tolerance through massive replication and “sloppy” elections are harder to see. LOCKSS as an application has many unusual characteristics. Nevertheless, we remain convinced that there is something fundamentally interesting in the idea of a system based on multicast protocols in which all actions are public and participants can make their own independent assessments of each other’s credibility.

One valid criticism of LOCKSS is that all monocultures are vulnerable, and if deployed *en masse* LOCKSS would be a monoculture. A bug in the implementation could wipe out information system-wide. It would be very valuable to have multiple independent implementations of the LCAP protocol. We hope that by keeping the protocol very simple we will encourage other implementations.

Acknowledgments

Grateful thanks for consistent support are due to Michael Keller, the Stanford University Librarian, to Michael Lesk at the NSF for funding the project with grant IIS-9907296, and to Sun Microsytems Laboratories, which has provided both time and funds.

The project was to a large extent inspired by Danny Hillis & Stewart Brand’s Millennium Clock project[clock].

Our thanks also go to the many people who reviewed the design and the paper, especially to Mark Seiden, Bob Sproull, the anonymous Usenix reviewers and Clem Cole; and to Michael Durket (Stanford University ITSS), Tony Smith-Grieco and Demian Harvill (Highwire Press) for help with implementation.

We're very grateful to our long-suffering alpha sites, and to AAAS, who allowed the valuable content of *Science Online* to act as the experimental subject.

References

[lamport] "The implementation of reliable distributed multiprocess systems" *Computer Networks* **2,** 1978. Butler Lampson provides a useful exegesis "How to build a highly available system using consensus" at http://www.research.microsoft.com/lampson/58-Consensus\Abstract.html.

[highwire] Highwire Press is at http://highwire.stanford.edu. A free example of their work is the British Medical Journal at http://www.bmj.com.

[bit-rot] Stanford's Conservation Online project maintains a page on this problem at http://palimpsest.stanford.edu/bytopic/electronic-records/electronic-storage-media/.

[format] Howard Besser maintains a website on this problem at http://sunsite.berkeley.edu/Longevity/. The NSF held a March 1999 workshop on it. See http://cecssrv1.cecs.missouri.edu/NSFWorkshop/execsum.html.

[alexandria] The Encyclopedia Britannica's article on the Library of Alexandria is at http://www.eb.com:180/bol/topic?eu=5704&sctn=1#s_top.

[lockss] The LOCKSS project website is at http://lockss.stanford.edu.

[disk-cost] "The price per megabyte [of disk storage] has declined at 5% per *quarter* for more than twenty years." Clay Christensen, *The Innovators Dilemma* (1997 Harvard Business School Press).

[w3mir] w3mir is supported by Nicolai Langfeldt at http://www.math.uio.no/~janl/w3mir/.

[apache] The Apache Software Foundation is at http://www.apache.org/.

[srm] Floyd, S., Jacobson, V., *et al*, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", IEEE/ACM Transactions on Networking, December 1997, Volume 5, Number 6, pp. 784-803. See http://www-nrg.ee.lbl.gov/floyd/srm-paper.html.

[enemies] 1997 was a bad year for libraries - see http://www.eb.com:180/bol/topic?eu=124351&sctn=1#s_top. Education efforts on book mutilation include http://gort.ucsd.edu/preseduc/bmlmutil.htm.

[slp] SLP is specified by RFC2165 at http://www.ietf.org/rfc/rfc2165.txt.

[ultima] The FAQs on the Ultima Online reputation system are at http://update.uo.com/repfaq/.

[digest] SHA-1 is specified by FIPS180-1 at http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[lrp] The Linux Router Project is at http://www.linuxrouter.org.

[castro] Castro, M. & Liskov, B. "Practical Byzantine Fault Tolerance", Proc. 3rd Symp. On Operating System Design and Implementation, New Orleans, Feb 1999. http://www.pmg.lcs.mit.edu/~castro/osdi99_html/osdi99.html.

[archive] The Internet Archive is at http://www.archive.org.

[nec] Chen, Y., Edler, J., *et al*, "A Prototype Implementation of Archival Intermemory", Tech. Rept. CEGGSY98, NEC Research Institute, Princeton NJ, Dec. 1998. See http://www.intermemory.org.

[dlr] The DLI2 initiative is at http://www.dli2.nsf.gov/.

[words] Phelps, T. A. & Wilensky, R. *Robust Hyperlinks Cost Just Five Words Each* is at http://HTTP.CS.Berkeley.EDU/~wilensky/robust-hyperlinks.html.

[freenet] **Clarke, I.,** *A Distributed Decentralized Information Storage and Retrieval System* is at http://freenet.sourceforge.net/Freenet.ps.

[napster] The Napster service is described at http://www.napster.com. *Wired* describes the controversial launch of Gnutella at http://www.wired.com/news/technology/0,1282,34978,00.html.

[**clock**] The Millennium Clock is a project of the Long Now Foundation at http://longnow.org/.

Remember to follow these links before they go 404!