M.I.T. DEPARTMENT OF EECS

6.033 - Computer System Engineering

Handout 27, April 8, 1999
Revision of April 10, 1999, 1:30 p.m.

Design Project #2: Maintaining the Integrity of a Digital Library

First, this is a **team project,** so form a team of 3 people where everyone has the same recitation instructor. Please send email to your recitation instructor before **Thursday, April 15** with the names and email addresses of your team members. Then get to work! Your team's report is due in recitation on **Thursday, April 29.** Complete details on turning in design project reports will be described in a future handout.

Motivation

Cheap disk storage now makes it possible to keep an entire library on line, for easy accessibility over the Internet. But as the quantity of material grows, it is important to consider methods of maintaining integrity of the data. Traditional methods of backup, such as making a copy to a tape each week, may not scale up to the sizes needed, and are not particularly well suited to the very slow rate of change of the data in a library.

I. The Design Problem

The Library of Congress has decided to digitize all of its 105,000,000 objects and store them on line with an array of low-cost hard disks. The objects vary in size from a few bytes to several Gigabytes, but they have an average size of 1 Megabyte each. Using the latest Seagate 50 Gigabyte disks, the Library would need to purchase an array of about 2,000 disks to hold all of this material. A disk array that large would be stretching things for a single computer system, but since disk sizes double roughly every 18 months, by the time they get finished digitizing everything in 2005, they figure that only 125 disks will be needed; this is well within the size range for a single large server. They call a system that holds a complete copy of this data a *replica site*.

The real problem that the Library of Congress is concerned with is maintaining the integrity of that many bits. To help ensure that a hurricane, meteor strike, earthquake, or civil war followed by NATO bombing doesn't destroy the entire collection, they have contracted with a set of widely separated site operators---in Sydney (Australia), Novosibirsk (Russia), Capetown (South Africa), and Quito (Ecuador)---each to install appropriate hardware and maintain a replica site holding the entire collection, in addition to the Library's own replica site in Washington, D.C. They are satisfied that the probability of all the replica sites being destroyed at the same time is negligible.

The set of replica sites, plus an appropriate set of protocols and procedures, becomes the *Repository*---an archival storage system---for the digital library. Other library functions, such as creating bibliographic records and searching those records or searching full-text indexes of the library contents, are handled by other systems; those systems use the Repository by storing or retrieving objects by object identifier via the Internet.

The Library has asked your team to design the protocols and procedures to add objects to the Repository and to update and maintain the replica sites, giving consideration to at least the following specific problems:

1. If a site is destroyed (e.g., a meteor takes out Capetown) the plan is to set up a replica at another site, say in the new U.S. Embassy in Nairobi. But the Library management is very concerned that any procedure that attempts to copy 105,000,000 files is likely to be interrupted a few times before it completes, and they want some assurance that the new replica will actually be complete.
2. Magnetic disks are good but not perfect--data written to disk sometimes comes back with errors or can't be read at all today, even though it was perfectly readable yesterday. The Library has a goal that if a file develops errors at some replica site it will be detected and repaired with high probability before the same file becomes unreadable at any other replica site; it will be detected with near certainty before the same file becomes unreadable at two other replica sites.
3. A team of librarians adds new things to the Library at the rate of about 1,500,000 per year, but they never remove anything and they never change anything already in the collection. When something is added to the collection, it is *not* necessary that all the replica sites be instantly updated, but it *is* necessary that they all eventually receive a copy, where "eventually" means within a few days. And it is important that once a librarian believes that the Repository has accepted responsibility for storing it, a new object not be lost. (The Library of Congress is running out of shelf space and is considering the adoption of a policy of shredding and discarding everything that has been digitized after it is placed in the Repository.) Notice that there may be some interaction between additions and creating a new replica: additions to the Repository will be ongoing during the time it takes to create the replica.

II. Underlying File System

You may assume that someone else has designed a simple file system that can gracefully deal with a very large number of files (the typical UNIX implementation would have some trouble keeping track of 105,000,000 files). Each replica site runs a copy of the file system. The file system provides exactly one directory per disk, each with a fixed name from the series /disk001/, /disk002/, etc. Below that the naming system is flat, which means that all of the files stored on the first disk are in the single directory named /disk001/.

This file system has the following methods in its interface:

    status = create-file(directory, metadata)

This creates a file consisting of the given metadata structure, {filename, size in bytes} *and* creates an empty data space of the given size.
status will be non-zero if there is not enough room for the file on this disk.

    write-file(directory, filename, data)

where data is assumed to be an array of bytes of the size that was specified in the metadata at create-file time.

    result = read-file(directory, filename)

where result is a structure consisting of two components: {file data, file metadata}.

    remove-file(directory, filename, data)

result = list-files(directory)
where result is a structure of two components:
{number of files in directory, array with one file name per element}

When the file system encounters a hard error (that is, an error for which retries don't help) on a disk, it flags that disk sector so that it will never be used again and it updates its record of the file contents to omit that sector. As a result, a read-file operation may deliver data that is missing one or more sectors, and a write-file operation may store an incomplete file. The file system uses files to store its own directories, but it maintains enough redundancy in its directory structure to reliably detect damaged or missing directory information. Thus it never returns or uses a partial or otherwise erroneous directory entry; the worst thing that happens is that one or more directory entries silently vanish.

Note that the file system delivers no warnings of missing data, incomplete writes, or vanishing directory entries. The reason is that, even though it may know about bad disk sectors, there are other failure cases that it cannot reliably detect. It is possible, for example, that an unnoticed hardware or software failure has led the file system to read the wrong set of disk tracks, or that the file system itself may have a bug and deliver data to its client in the wrong order. Using an end-to-end argument, its designers decided it was better to leave the problem of detecting errors in client data entirely to the client of the file system.

The file system is designed to work correctly with multiple threads. Each of these methods is *isolated* (that is, atomic with respect to parallel threads that invoke the same or other file system methods). In addition, create-file and remove-file are *recoverable*. That is, following a crash all directories are intact and a file that was in the midst of being created or removed will either be intact or gone. Write-file is *not* recoverable; the file system makes no attempt to reconstruct or recover file contents following a crash. On the other hand, crashes do not affect files that were not being written at the moment of the crash. The file system does not come with any backup or logging features, and there is no provision for security (that is, no access control lists, owners, or permission bits). The metadata of the file system is configurable, so if you would like to have the file system store any additional metadata, you should specify carefully what you need. You may discover other requirements or features for the file system. If they seem consistent with the general design philosophy so far, you can request that the file system designers add those features or meet those requirements.

III. Other Considerations

One property of large research libraries should be kept in mind: much of the material in such a library is rarely used! It is typical to find that 50% of the objects collected by a really big library are not requested by anyone during the first 100 years they are in the collection, and that 95% of the objects are used less than once a decade. (One might try to exploit this usage pattern to keep rarely used material on lower-cost, slower storage, but that is *not* the objective of this design project.)

IV. Summary of Your Task

As indicated above, your team's job is to design the Repository. (Section VI offers some suggestions for writing and organizing your writeup.) The repository is to be used as a component of a larger system in which readers search a catalogs or indexes or otherwise discover references that consist of object identifiers, and they will present this object identifier to the get method of the Repository. Librarians will assemble a collection of bits that repesents a new object, and they will add that object to the Repository by invoking the put method.

Thus as part of the overall report, your design should include discussion supported by pseudocode (which can include, for example, actions by human operators) for at least the following specific procedures:

- Reading data from the Repository:
  data = get(object-id)
  (Assuming your design is successful, the only error that will ever occur is *"no such object-id"*.)
- Writing data into the Repository:
  object-id = put(data)
  (Note that the Repository tells the writer the id of the newly installed object.)

Also include a discussion, supported by pseudocode, for the following activities:

- Ridding the Repository of bogus and duplicate data.
- Ensuring continued integrity of stored data.
- Creating a new replica site in the case of a massive site failure.

Although the Repository clearly requires some security features to ensure that only librarians can add objects, while anyone in the general public can perform gets, you are not expected to design that part of the Repository.

V. Suggestions (not mandatory, but you may find useful ideas here)

1. Read chapter 8, which will be handed out next week in lecture. It contains concepts that apply to the design and terminology that you should use in your report.
2. Track down some information about typical modern disk speeds and failure rates in the field, and use that information to make reasonable estimates of the time it will take to do various operations and how often they will need to be done. (You may assume that failure rates five years from now will be similar to failure rates today. Also make sure to provide references to where you found your information.)
3. Do an error event analysis, as described in an upcoming lecture, for various layers of your system, dividing the possible events into detectable, anticipated, and intolerable errors. (The glossary for chapter 8 gives definitions of these terms.) Start by performing such an analysis on the raw materials you have to work with---the file system, the network, and the replica sites. Then, as you develop your design, create a similar analysis for the Repository as a whole.
4. Check out some of the papers in the readings *before* they are assigned for classroom discussion. Specifically, you will find that the three papers on **RAID (reading 27), System R (reading 29),** and **LFS (reading 30)** contain ideas and ways of thinking that you may be able to use in your design. If you postpone reading them to the night before they are discussed in class, it will be too late to help in your design. **But be wary of trying to adopt directly the solutions those papers describe.** For example, RAID provides very little help for a site disaster, but the motivation that underlies RAID is very similar to the motivation that underlies having replica sites. (And in any case, that paper explains how disks work.)
5. Consider carefully your strategy for naming files in the Repository, and for mapping the externally used object identifiers to the file names you use at each replica site. Object identifiers will be stored in catalogs and indexes of other systems, and you don't want to have to update all of those

catalogs and indexes every time you move a file to a different disk in one replica. Also, consider carefully whether or not it is important to your design that a given object be stored using the same file name in each replica.

6. Simplicity is crucial to the correct and reliable operation of such a large-scale system, so look for algorithms that accomplish several things at the same time. For example, if you develop a procedure for keeping replicas consistent despite occasional files being lost to decay, you may be able to simplify the job of creating a new replica by first creating an approximate replica and then letting the consistency-maintaining system finish the job.
7. If you need a high-quality checksum, consider the use of MD5, which produces a 128-bit cryptographic hash of its input.
8. If you need space in primary memory for buffers, indexes, etc., you can assume that several Gbytes are affordable (figure $500/Gbyte in 2005) and that the computer system has enough address space to allow attaching that much primary memory. But keep in mind that primary memory is volatile: on a system crash, it all resets to zero.
9. Your design may benefit from being organized in two layers. The upper layer consists of operations on the Repository; the lower layer consists of operations on a single replica site. The lower-layer operations are in turn built on the methods of the file system. If you do this, you may want to also do a separate error event analysis of the layers.
10. When considering integrity maintenance, it may be useful to modularly separate three activities: (i) discovering damaged files; (ii) discovering lost files; and (iii) copying files from one replica to another to repair damage or loss.

VI. Suggestions on Writing Style

Your paper should be as long as is necessary to explain the problem, your solution, the alternatives you considered, and the reasons for your choices. It should be no longer than that. We estimate that a good paper will be 8 to 10 pages in length.

A good paper begins with an abstract. The abstract is a very short summary of the entire paper. It is *not* an outline of the organization of the paper! It states the problem to be addressed (in one sentence). It states the essential points of your solution, without any detailed justification. And it announces any conclusions you have drawn. Good abstracts can fit in 100-150 words, at most.

The body of your paper should expand the points made in the abstract. Here you should:

1. Explain the problem and the externally imposed constraints.

You should explain to your intended audience the background of the problem in terms that the audience can understand.

2. Explain and elaborate your solution.

Be sure to explain the approach or architecture conceptually *before* delving into details, components, and mechanics. (Remember, you aren't writing a mystery novel!)

3. Show how your solution satisfies the constraints and solves the problem (or how it fails to do so);

Explain how the properties of your solution that result from choices you have made in your design are reasonable or desirable in the context of the problem statement. Include analysis of the estimated (or measured, if it applies) performance characteristics of your solution.

4. Describe the alternative approaches that you considered and rejected, and why you rejected them.

Your paper will be more convincing if you say not just why your idea is good, but why it is better than the alternatives. (For example, if another approach would meet all of the objectives perfectly, but the cost would be 100 times higher, then you should mention that as a reason for choosing your less general but cheaper approach.)

5. Document your sources, giving a list of all references (including personal communications). The style of your citations (references) and bibliography should be similar to the styles in the technical papers you're reading in this class. In particular, a bibliography at the end and no citations in the text of your paper is insufficient; we'd like to see what specific pieces of information you learned from where as we read your paper.

Write for an audience consisting of colleagues who took 6.033 five years ago. That is, they understand the underlying system and network concepts and have a fair amount of experience applying them in various situations, but they have not thought carefully about the particular problem you are dealing with. Assume that your paper will also be used to convince the Library staff that you have a viable design. Finally, give enough detail that they can turn the project over to their Information Technology programmers for implementation with some confidence that you won't surprised by the result.

VII. How do we evaluate your paper?

When evaluating your report, your instructor will look at both content and writing.

Some content considerations:

- Does your solution actually address the stated problem?
- Do you explain your decisions and the trade-offs?
- How complex is your solution? Simple is better, yet sometimes simple won't do the job. But unnecessary complexity is bad.
- Does your solution fit well with the rest of the system? If your solution requires modifying every piece of hardware, software, and data in sight, it won't be credible, unless you can come up with a very good story why everything needs to be changed.
- How extensible is your design? Are there opportunities for later addition of desirable features that you decided to omit?

Some writing considerations:

- Is the report easy to comprehend?
- Is it well organized and coherent?
- Does it use diagrams where appropriate? (A frequent problem when people use word processors is that they try to express everything in words, either because the word processor doesn't make it easy to include diagrams, or they haven't ever learned how to use the drawing features. Pictures can communicate some ideas far better.)
- Does it use the concepts, models, and terminology introduced in 6.033? If not, does it have a good

reason for using a different universe of discourse?
- Does it address the intended audience?
- Is there a good abstract and bibliography?

You can find other helpful suggestions on writing this kind of report in the M.I.T. Writing Program's on-line guide to writing Design and Feasibility Reports. Another useful reference is the following book: "The Elements of Style," by William Strunk Jr. and E. B. White, Third Ed., MacMillan Publishing Co., New York, NY, 1979. There are other good books on technical writing style too.

VIII. Revision History

- 4/8/99 Restored missing spaces in text of file system specification.
- 4/9/99 Removed bogus status return value on specification of write-file.
- 4/9/99 "repository" changed to "replica" in suggestion 10.
