ARCHIVAL REPOSITORIES FOR DIGITAL LIBRARIES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arturo Crespo
March 2003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andreas Paepcke

Approved for the University Committee on Graduate Studies:

Abstract

Current libraries use an assortment of uncoordinated and unreliable techniques for storing and managing their digital information. Digital information can be lost for a variety of reasons: magnetic decay, format and device obsolescence, human or system error, among many others. In this thesis, we address the problem of how to build *archival repositories* (AR). An AR has the following combined key requirements that distinguish it from other repositories. First, digital objects (e.g., documents, technical reports, movies) must be preserved indefinitely, as technologies, file formats, and organizations evolve. Second, ARs will be formed by a confederation of independent organizations. Third, published digital objects have a historical nature, so changes should not be done in-place; instead, they should be recorded in versions. We provide an architecture for ARs that assures long-term archival storage of digital objects. This assurance guarantee is achieved by having a federation of independent but collaborating sites, each managing a collection of digital objects. We also provide a framework for evaluating the reliability and cost of an AR. Finally, we present techniques for efficient access of documents in a federation of independent sites.

Acknowledgments

First, I would like to thank my advisor, Hector Garcia-Molina who taught me much of what I know about research, clear thinking, and effective writing. Working with Hector was not only a privilege but a pleasure as well. I also want to thank the other members of my reading committee, Andreas Papecke and Jennifer Widom, as well as my orals committee members, Chris Jacobs and Terry Winograd, for their many useful suggestions and comments.

I would like to thank Jerry Saltzer for providing the initial spark that ignited my interest on this topic. In addition I thank Carl Lagoze for his useful suggestions and encouragement on pursuing this topic.

I would also like to thank all the members of the Stanford Database Group for their helpful advice, insightful discussions, and for always being there during good and bad times. In particular, I would like to thank my co-authors Orkut Buyukkokten and Brian Cooper as well as Stefan Decker (whose ideas lead to the work presented in Chapter 7). Finally, a special thank to my officemates Sergey Brin, Wilburt Labio, Claire Cui, Taher Haveliwala, and Glen Jeh, who made the long hours at Stanford much more enjoyable.

At Stanford, I was fortunate to have met, and count among my friends, a number of smart, interesting, and caring people. The list of friends that provided me support and encouragement is far too long to write here, but I would like to mention a couple of names. Very special thanks to Erik Boman and the leaders of the Stanford Outing Club, who introduce me to the natural wonders of California; and to Carlos Guestrin for encouraging me to break away with my self-imposed limits allowing me to to go farther and higher than I ever thought I could ever go. I also want to thank

Sergey Melnik not only for his true friendship, but also for his technical help in developing some of the ideas in this dissertation.

Finally, I want to thank my family: my parents Arturo and Inmaculada, my sister Massiel, and my brother Joaquin. Familia, esta tesis se la dedico a ustedes. Sin su fe en mi y su apoyo incondicional escribir esta tesis hubiese sido imposible.

Contents

List of Figures

Chapter 1

Introduction

The advancement of human knowledge critically depends on the preservation of knowledge gained in the past. Currently, more and more of that knowledge is stored electronically, in the form of files, databases, web pages, and programs. Unfortunately, most of the systems that store this information are not designed for preserving information over very long periods of time, and their information is vulnerable as technologies and organizations evolve. Furthermore, the information is often stored in an uncoordinated and unintegrated fashion. This can lead to the loss of important components of the intellectual record. This problem will only get worse as more and more information is provided only in digital form.

Digital information can be lost in many different ways and in Chapter 2, we will study in detail the threats to digital information. Some of the major threats to digital information are:

Media decay and failure: Natural decay processes, such as magnetic decay, may make media unreadable. For reference, while microfilm has a life of 100 to 200 years, magnetic tapes decay in only 10 to 20 years. Media may also fail when it is being accessed, e.g., a tape may break when in a reader, or a disk head may crash into a platter.

Access Component Obsolescence: A document may be lost if we do not have one of the components necessary for supporting access to it. For example, components required to access documents stored in a floppy disk include a floppy drive and the drivers to access it. If we have a document in a 8 inch floppy disk, but we do not have a 8 inch floppy drive, we will not be able to access the document (even if the media is still readable).

Human and Software Errors: Documents can be damaged by human errors or by software bugs. For example, a person or a program may delete a document (e.g., by reformatting the disk that holds the document), or may improperly modify the files and data structures that represent the document (e.g., by writing random characters in the middle of the document).

External Events: Events external to the archival repository, such as fires, earthquakes and wars, may of course also cause damage.

When considering long-term preservation of digital objects, there are two interrelated factors: data and meaning preservation. To illustrate, consider the Mayan inscriptions on their temples. For us to “read” them, first the carvings and paintings had to be preserved (data preservation) over the centuries. Second, the meaning of their hieroglyphs had to be decoded. Thus, to preserve the meaning there needs to be some translation machinery, which is based on a lot of guesswork (as in the case of Mayan writings), or aids left behind (which are of course extremely hard to provide in advance). The translation could be done gradually and continuously, to avoid spanning large differences in representations (e.g., translating a document in MS Word 4 to Word 5 to Word 6).

In this dissertation we focus on data preservation only. This is admittedly the much simpler of the two problems, but clearly, without data preservation as a first step, meaning cannot be preserved. Thus, we view a *digital object* as a bag of bits (with some simple header information, to be discussed). We will not concern ourselves here on whether this object is a postscript file (or any other format), the document that explains how postscript is interpreted (an aid for preserving the meaning of the postscript file), or an object giving the metadata for the postscript file (e.g., author, title). However, we do wish to preserve *relationships* among objects. That is, we will develop an identification scheme so that one object can “point” or “reference” another one. This way, for instance, the metadata object we just discussed can identify the postscript file it is describing.

Our approach to dealing with the problem of data preservation is to build a reliable archival repository that protects digital information from failures. Users who create digital documents would deposit these documents in their local repository. This repository would then take whatever actions were necessary to protect the documents. Our vision of an archiving system is built on several principles:

- Write-once archiving: Documents, once archived, are never modified or deleted. Updates to documents are represented as version chains.

- Federated replication: A federation of archives work together, storing copies of each others' data so that if a site fails, no documents are permanently lost.

- Scientific design: The policies for constructing and operating an archiving system should be studied scientifically, to find the most reliable and resource efficient techniques.

- Robust operation: The system should be robust in the face of failures. This means both that documents are preserved, and also that the system operations (document replication, content searches, and so on) continue to run.

Building an archival repository is a complex task because the problem of long-term preservation entails not only a large number of uncertainties about the future, but also a large number of configuration options that could impact the reliability of the system. For example, how many copies should we make of each document? How frequently should we check these copies for corruption? Should these copies be stored on a few expensive disks or many cheap disks? In addition, the designer needs to predict future events such as the reliability of sites and disks, survivability of formats, how many resources will be consumed by the recovery algorithms, how frequently the recovery algorithms will be invoked, how many user accesses will be made to the documents, and many other uncertainties. In addition to the challenges of long-term preservation, an archival repository is only useful if the documents can be accessed and found efficiently.

Given our problem definition, the reader may wonder if data preservation is a solved problem. After all, a database system can very reliably store objects and their relationships. This may be true, as long as the same or compatible software is used to manage the objects, but it is not true otherwise. For instance, suppose that the Stanford and MIT libraries wish to store backup copies of each other's technical reports, but they each use different database systems. It is not possible (at least with current systems) to tell the Stanford system that an object is managed jointly with MIT. Similarly, if Stanford's database vendor goes out of business in 500 years, or Stanford decides to use another vendor, migrating the objects elsewhere can be problematic, since database systems typically represent reliable objects in ways that are intimately tied to their architecture and software.

The objective of our work is *not* to replace database systems, but rather to allow existing and future systems to work together in preserving an interrelated collection of digital objects (and their versions) in the simplest and the most reliable possible way. We aim for a system that provides preservation, even if some efficiency is lost, while ensuring the autonomy of individual archives. Our work complements other investigators' in areas such as preserving digital formats, ensuring the security of digital objects against malicious users, encoding semantic meaning in digital documents, and so on.

This dissertation is divided into three main parts: (i) design and evaluation of ARs (Chapters 2 and 3), (ii) a reference architecture and algorithms for ARs (Chapters 4 and 5), and (iii) efficient access of ARs (Chapters 6 and 7). Relevant related work is covered in each chapter of the dissertation. Specifically, in this dissertation we study:

- *AR Reliability Analysis* (Chapter 2): In this chapter we study the *archival problem*, how a digital library can preserve electronic documents over long periods of time. We analyze how an archival repository can fail and we present different strategies to help solve the problem. We introduce *ArchSim*, a simulation tool for evaluating an implementation of an archival repository system and compare options such as different disk reliabilities, error detection and correction algorithms, and preventive maintenance.

- *AR Design Methodology* (Chapter 3): Building on the evaluation tools presented in Chapter 2, we study the design of archival repositories. Designing an archival repository is a complex task because there are many alternative configurations, each with different reliability levels and costs. The objective of this chapter is to be able to answer questions such as: how much would it cost to have an AR with a given archival guarantee? Or the converse: given a budget, what is the best archival guarantee that we can achieve?. We achieve this objective by systematically and scientifically analyzing each step of the design with the aid of an ArchSim extension that models and evaluates costs. In summary, in this chapter, we study the costs involved in an Archival Repository and we introduce a design framework for evaluating alternatives and choosing the best configuration in terms of cost. The design framework and the usage of the ArchSim extension are illustrated with a case study of a hypothetical (yet realistic) archival repository shared between two universities.

- *Cellular Repository Architecture* (Chapter 4): In this chapter we present a reference architecture for archival repositories. This architecture is formed by a federation of independent but collaborating sites, each managing a collection of digital objects. The architecture is based on the following key components: use of signatures as object handles, no deletions of digital objects, functional layering of services, the presence of an awareness service in all layers, and use of disposable auxiliary structures. Long-term persistence of digital objects is achieved by creating replicas at several sites.

- *Awareness Algorithms* (Chapter 5): One of the most critical components in an AR is the awareness mechanism, used to notify clients of inserted, deleted or changed objects. In this chapter we survey the various awareness schemes (including snapshot, timestamp and log based), describing them all as variations of a single unified scheme. This makes it possible to understand their relative differences and strengths. In particular we focus on a signature-based awareness scheme that is especially well suited for digital libraries, and show enhancements to improve its performance.

- *Routing Indices* (Chapter 6): Finding information in a large federation of ARs, such as the one proposed in Chapter 4, currently requires either a costly and vulnerable central index, or flooding the network with queries. In this chapter we introduce the concept of Routing Indices (RIs), which allow ARs to forward queries to neighbors that are more likely to have answers. If an AR cannot answer a query, it forwards the query to a subset of its neighbors, based on its local RI, rather than by selecting neighbors at random or by flooding the network by forwarding the query to all neighbors. We present three RI schemes: compound, hop-count, and exponential. We evaluate their performance via simulations, and find that RIs can improve performance by one or two orders of magnitude vs. a flooding-based system, and by up to 100% vs. a random forwarding system. We also discuss the tradeoffs between the different RI schemes and highlight the effects of key design variables on system performance.

- *Semantic Overlay Networks* (Chapter 7): Current network topologies for large federations of nodes, such as the ones used in P2P networks, are usually generated randomly. Each node typically connects to a small set of random nodes (their neighbors), and queries are propagated along these connections. Such query flooding tends to be very expensive. We propose that node connections be influenced by content, so that for example, nodes having many "computer science" documents will connect to other similar nodes. Thus, semantically related nodes form a Semantic Overlay Network (SON). Queries are routed to the appropriate SONs, increasing the chance that matching files will be found quickly, and reducing the search load on nodes that have unrelated content.

We conclude this thesis by discussion challenging open problems in Chapter 8.

Part I

Design and Evaluation of Archival Repositories

Chapter 2

Evaluating the Reliability of an AR

2.1 Overview

One of the main challenges in designing an archival repository (AR) is how to configure the repository to achieve some target "preservation guarantee" while minimizing the cost and effort involved in running the repository. For example, the AR designer may have to decide how many computer sites to use, what types of disks or tape units to use, what and how many formats to store documents in, how frequently to check existing documents for errors, what strategy to use for error recovery, how often to migrate documents to a more modern format, and so on. Each AR configuration leads to different levels of assurance; e.g., on the average a document will not be lost for 1000 years, or in 1000 years we expect to still have access to 99% of our documents. Each configuration has an associated cost, e.g., disk hardware involved, computer cycles used to check for errors, or staff running each site.

The number of options and choices is daunting, and the AR designer has few good tools to help in this task. The traditional fault tolerance models and techniques, of the type used to evaluate hardware, are a helpful starting point, but they do not capture the unique complexities of ARs. For example, traditional models may have difficulty capturing different document loss scenarios (e.g., missing interpreter, missing bits, missing metadata) and they frequently assume failure distributions (e.g., exponential) that are too simplistic.

In this chapter we present a powerful modeling and simulation tool, *ArchSim*, for helping in AR design. ArchSim can model important details, such as multiple formats, preventive maintenance, and realistic failure distribution functions. ArchSim is capable of evaluating a large number of components over very long time periods. ArchSim uses specialized techniques in order to run comprehensive simulations in a time frame that allows the exploration and testing of different policies.

Of course, no model can be absolutely complete: there is an intrinsic tradeoff between how detailed the model is and the complexity (even feasibility) of its analysis. In our case, we have chosen to ignore (at least in this initial study) information loss due to format conversion (migration to a new format is always successful and does not introduce any loss). We also do not model partial failures (a failure in which we can salvage part, but not all, of the information in a device). As we will see, we also rely on an "expert" that can provide failure distributions for the components of the systems. For instance, if format obsolescence is an issue, an expert needs to give us the probabilities of different format lifetimes.

In summary, in this chapter we present:

- A comprehensive model for an AR, including options for the most common recovery and preventive maintenance techniques (Sections 2.2, 2.3 and 2.4).
- A powerful simulation tool, ArchSim, for evaluating ARs and for studying available archival strategies (Section 2.5).
- A detailed case study for a hypothetical Technical Report repository operated between two universities. Through this case study, we evaluate AR factors such as disk reliability, handling of format failures, and preventive maintenance.

2.2 Archival Problems and Solutions

We define an *Archival Repository* (AR) as a data store that gives a guarantee of long-term survivability of its collection. We will refer to each unit of information in the AR as an *Archival Document* or just *document* for short.

As a first step in modeling and evaluating ARs, it is important to understand how information can be lost, and what techniques can reduce the likelihood of loss.

2.2.1 Sources of Failures

An AR fails to meet its guarantee when the archive loses information. Such a loss may be caused by a variety of undesired events, such as the failure of a disk, or an operator error. A document is lost if the bits that represent it are lost, and also if the necessary components that give meaning to those bits are lost. We define the *components* of a document to be all the resources that are needed to support access to the document, e.g., the document bits, the disk that stores the bits, and the viewer that interprets the bits.

An undesired event does not necessarily cause information loss. For instance, if the AR keeps two copies of a document, and the disk holding one of the copies fails, then the document is not lost. It would take a second undesired event affecting the second copy to cause information loss. A document is *damaged* if a copy or instance of one of its components is corrupted or lost. Damaged documents should be *repaired* by the system to protect them from further failures.

The most common undesired events that may lead to the loss of a document can be broken down into the following categories. (We do not consider transient failures, e.g., a power failure, that do not lead to a permanent loss.)

Media decay and failure: Natural decay processes may make media unreadable. A well known example is magnetic decay, occurring when media loses its ability to hold the "bits" that encode a document. In Figure 2.1, we summarize typical decay times for electronic media [8]. These decay times assume "good" storage and operating conditions, and infrequent accesses. Note that decay typically does not affect all parts of a tape or disk simultaneously. For example, a few bits may first decay, rendering a document (or part of a document unreadable), followed by other decays. Figure 1 gives times to the first decay.

Media	Decay Time
Magnetic Tapes	10-20 years
CD-ROM	5-50 years
Newspaper	10-20 years
Microfilm	100-200 years

Figure 2.1: Typical Media Decay Times

Media may also fail when it is being accessed; e.g., a tape may break when in a reader, or a disk head may crash into a platter. The likelihood of failure increases with the frequency that the medium is accessed. For example, the expected life of a hard disk that starts and stops spinning very frequently is 5 years, while it is 10 to 20 years for an inactive disk.

Component Obsolescence: A document may be lost if we do not have one of the components necessary for supporting access to it. To illustrate, we describe two important forms of component obsolescence: *media obsolescence* and *format obsolescence*. Media obsolescence occurs when we do not have a machine that can read the medium (even if the medium is still readable). Format obsolescence happens when the system does not know how to interpret the bits that are encoded in the medium. This kind of failure is the result of the non-self-describing nature of information stored in electronic media. That is, the stored bits must be transformed into a human comprehensible format by a process that cannot be inferred from the original bits. For example, say we have a postscript document. To view it, we must (a) know that its format is some particular version of postscript, (b) have a program that can interpret that version of postscript and display the document on a screen (or printer), and (c) have a computer that can execute the code for the program. If we do not have any of these components, then we cannot access the Postscript document.

Human and Software Errors: Documents can be damaged by human errors or by software bugs. Reference [37] suggests that humans and software are the most serious sources of failures. Better technologies and designs are reducing media decay and component failure rates, but there is no sign of a drop in the rate of faults created by software and human errors.

External Events: Events external to the AR, such as fires, earthquakes and wars, may of course also cause damage. Such events may cause several AR components to fail simultaneously. For example, a flood can destroy a collection of disks at one site. If a document were replicated using those disks, all copies would be lost.

2.2.2 Component Failure Avoidance Techniques

There are two well-known ways to avoid an AR failure: we can either reduce the probability of component faults, or we can design the AR so those faults do not result in document loss. In this subsection, we review techniques in the first category, while the next subsection will cover the second category. We organize the presentation in this subsection using the taxonomy of faults presented in Section 2.2.1.

Media decay and failure

A fundamental decision in designing an AR is which medium will be used. In this subsection, we first outline how we can choose the appropriate medium for an AR. Then, we present strategies that allow reduction of the rate of decay and failure for these media.

The question of which medium to use involves two decisions. First, we need to decide on the access properties that will be needed from the medium. Second, we need to find the most cost-effective technology that provides these access properties. By access properties, we mean the characteristics of the medium such as how fast can we start reading the medium, how fast can we find a document, if we can rewrite a document without having to rewrite the whole medium, etc.

The reason we need to consider the access properties is that they will affect how we design our system. For example, the time that it takes to start accessing a medium will have an impact on the available strategies to reduce Media Decay. Specifically, on-line media (i.e., media with very little initial delay) allow for easy detection of decay, as it is cheap to access all the information periodically. On the other hand, detecting decay on off-line media is more complicated because it is expensive to load and read the media. Nevertheless, we need to recognize the tradeoffs between different kinds of media, as off-line media tend to have a lower cost of maintenance and a longer expected life than on-line media.

In terms of underlying technologies, there are many alternatives for archival media. In our work, we have concentrated on "digital" media; however, the boundary between digital media and paper has been diffused by some interesting new media that have been developed recently. For example, "PaperDisk" technology uses 2D barcodes to print electronic information on paper [2]. The barcodes can then be scanned back into the computer and transformed into digital information. By using special algorithms, this technology can save up to 1Mb of information on a sheet of paper.

For ARs, the most important factor in choosing media technology is its lifetime. Evaluating the lifetime of media technologies is a complicated process. Media lifetime is nearly impossible to determine in practice (we cannot wait "n" years for the medium to fail). Therefore, estimates are based on the physical properties of the medium. The challenge is to find which are the relevant properties (so we can find the weakest link) and what is the appropriate model of decay for those characteristics. For example, a CD-ROM is basically a reflective layer sandwiched between a clear substrate and a lacquer film. A CD-ROM can fail because of either the deterioration of the reflective layer or because of the loss of optical clarity of the substrate. Deterioration of the reflective layer can occur because of oxidation, corruption, or delamination. As these three sources of deterioration are well known for the materials used in the reflective layer, we can predict when they will deteriorate and eventually fail. Similarly, we can study the causes for losing optical clarity of the substrate and try to predict when this failure will occur.

Beyond the choice of which medium to use, we need to make certain that the medium is stored in the best conditions to ensure its preservation. In the case of tapes, a controlled environment with low temperature and low humidity may increase the life of the tape in an order of magnitude [9].

Component Obsolescence

Avoiding component obsolescence is a complex problem, as an accurate prediction of which operating systems, document formats, and devices will be used in the future is impossible. However, there are some techniques that help avoid the component obsolescence problem. In this subsection, we will briefly explore three of these techniques: usage of standards, self-contained media, and preservation of equipment. In the next section, we will study system techniques that can lessen the problem of component obsolescence.

One way to avoid the component obsolescence problem is to use standards. Standards are (usually) well-documented, which may allow the re-creation of readers for that standard. However, there are some problems with standards. First, there is a bootstrap problem: a person recreating a reader needs to be able to read the documentation for the standard. Second, an important risk when using a standard is that many documents that claim to follow a standard are not really 100% compliant. For example, most HTML documents on the web do not use HTML correctly, so a browser that is built to accept only correct HTML will not be able to display most documents.

Another solution to the obsolescence problem is the preservation of equipment. This approach has been attempted by the National Media Lab, but it is a difficult and costly task, especially when the parts needed to repair preserved equipment are no longer available.

Another interesting proposal for solving the problem of component obsolescence is the use of a self-contained medium such as the electronic tablet [42]. The electronic tablet is a portable computer that contains all the necessary hardware and software (as well as data) to make a document readable. The only needed input for this tablet is a power source. Although the electronic tablet also solves the component obsolescence problem (as everything needed to read the document is in the tablet), no tablets have been built and it is yet to be determined how cost efficient they will be.

Human/Software Errors

As stated before, human and software errors are the fastest growing source of faults in computer systems. Beyond traditional techniques such as program validation, and user training, some specific guidelines are useful in an AR. These guidelines include:

- *Define interfaces that minimize the amount of damage that can be done:* For example, by not allowing deletions or updates in the AR, users cannot make mistakes that will result in the deletion of a file. In addition, by defining a restricted interface, it is easy to distinguish between faults and the normal operation of the system. For instance, if a file disappears in a system that does not allow deletions, then we know that a fault has occurred.

- *Minimize the amount of code that can cause damage:* The assumption is that the smaller the amount of code, the better the chance that it can be extensively tested and verified.

- *Use hardware safeguards to prevent software errors:* Hardware can help prevent damage to documents. For example, by opening the “read-only tab” in a diskette, we instruct the hardware to disallow any writes to it. Another commonly used hardware protection is virtual memory that can be used to prevent unauthorized writes to some memory addresses.

External Events

A range of techniques can be used to prevent damage from external or environmental events. These techniques include, for example, fire-proof walls, earthquake-proof buildings, etc. These avoidance techniques are beyond the scope of this thesis and will not be discussed further.

2.2.3 System-Level Techniques

Migration

Migration involves replacing a document component with a new, safer one. For example, suppose that "Postscript" readers are becoming obsolete and are being replaced by new "Postscript II" readers. Then, we may decide to migrate Postscript documents into the new format before Postscript readers become unavailable. Migration is particularly effective for storage devices. However, migrating to new formats is more challenging because some information may be lost in the transformation.

Migration techniques can be classified in a continuum, ranging from passive strategies to active strategies. In Figure 2.2 we show the spectrum in the context of reducing media decay. In the figure, we can see that passive strategies include reducing the media decay time by either using better media or storing media in a "controlled" environment. Active strategies attempt to reduce media decay by reading and copying the data periodically. When the copies are made over the same medium, the strategy is called *periodic refreshing*. When the copies are made in a different medium, then the strategy is usually called *periodic migration*. There are many tradeoffs between active and passive strategies. In general, passive strategies have the advantage of lower use of resources than active strategies, but, unfortunately, they also provide a lower probability of preservation. Active strategies have a high probability of preservation, but they consume more resources. This last point is one of the reasons why archival models are needed; designers need to make strategy decisions that achieve their *target* archival guarantee while reducing the total amount of resources consumed.

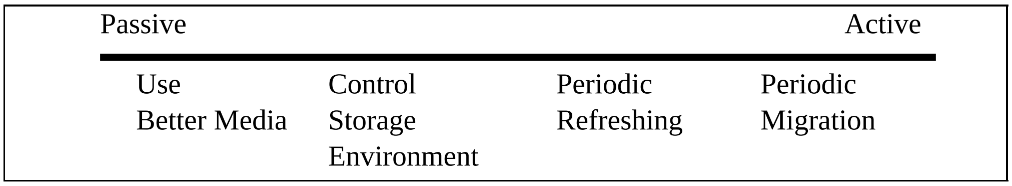

Figure 2.2: Reducing Media Decay

Finally, migration can also be used to avoid component obsolescence. However, performing a migration process that does not lose some information or formatting is frequently impossible (especially when used transitively, that is, when a document in a given format is migrated to another format, and then in turn is migrated into yet another format). For example, consider translating an ancient text from Greek to Latin, and then from Latin to English. Most people would agree that if the Greek version were available, a direct translation from Greek to English would be better.

Replication

Replication makes copies or creates new instances of needed components, to ensure the long term survival of the component. For example, to prevent media failure, we can write the document onto multiple tapes. If one of the tapes is damaged, then we can use the remaining tapes to access the document. Replication can also be used to survive component obsolescence. For example, a document can be written in multiple formats; in this way, if a format becomes obsolete, we can still access the document in another format.

Unfortunately, replication by itself does not improve the mean time to failure (MTTF) of a system. In [37], it is shown that without repairs the MTTF of a systems may *decrease* when using replication. Specifically, to obtain a benefit from replication, after a failure has been detected, we should repair it as rapidly as possible.

An additional advantage of replication is that it simplifies failure detection. If we keep three copies of a document in three different tapes and one of the copies is different, then we can assume, with very high probability, that the copy that is different has failed and should be discarded.

Replication is used by many archival system including the Computing Research Repository [34], the Archival Intermemory Project [28, 13], and the Stanford Archival Vault [17].

Emulation

Emulation involves recreating all the components needed to access a document on a new platform [70]. Emulation can be done at several levels: hardware, operating system, or application software. The computer game community has embraced emulation, and many "platform" emulators are available that allow old computer games to run on modern computers.

The main advantage of emulation is that it can potentially improve the scalability of archival systems. This scalability improvement is the result of emulation recognizing that components are frequently stacked on top of each other (e.g., to read a document, we need to run software, which needs an Operating System, which in turn needs some specific hardware). By choosing an appropriate level, let us say the Operating System level, we do not need to worry about components higher than this level. In other words, if we produce an emulator of Windows 98, we do not need to do anything special in order to use all the programs currently available for that platform. (On the contrary, with all other system solutions, we would need to concern ourselves with each possible component independently of each other.) In addition to this benefit, emulation better preserves the "look and feel" of the original application (which may be important in some fields).

One major disadvantage of emulation is its feasibility. Emulating a piece of software/hardware can be a tricky and difficult task that requires a lot of engineering and collaboration from the current producers of the component. Additionally, the task may infringe on industrial property and copyright laws. In addition, the benefits of emulation may not materialize in many situations. For example, if we have a relatively uniform collection of documents, the cost of producing an emulation system will probably be much higher than the cost of using replication. In addition, preserving the "look and feel," while important in some contexts, can be negative in others (e.g, we do not want to preserve the "feeling" of waiting a couple of days for the answer to a query when accessing an archival database).

Fast Failure Detection and Repair

With most system fault tolerance techniques, we need to check periodically for failed documents. Fast failure detection and repair yields improved reliability. For example, if one of two component copies have failed, the sooner we detect the problem and generate a new copy, the more protection we have against a second fault.

In general, it is impractical (and sometimes impossible) to detect exactly when a fault in a component has occurred. This is because the manifestation of the fault, an error, might only appear long after the fault occurred. For example, a tape can deteriorate and lose information (a fault), but we will not know that the tape has failed until we attempt to read the data and we get an error. Testing for errors may be a difficult task. A typical system has a very large number of components and it would take a long time to check all of them for errors. In order to reduce the time spent in finding errors, we can use sampling (by time or component type). We can also use partial checks or predictors to select the components that have a high probability of failure.

- *Time-based samples:* Documents are probed at specific intervals of time so the probability of losing a document due to multiple failures is small. Obviously, the challenge of this technique is finding the appropriate time interval. Later, in the experiments section of this chapter, we will see how can we use simulations to find good detection intervals.

- *Component-based samples:* This technique probes components instead of individual documents, an especially useful approach when the number of components is much smaller than the number of documents. Specifically, the system will perform a test on the component, under the assumption that if the component passes the test, it is most likely that the component will successfully work for all documents. A way to perform the test is to select one document (or some small number of documents) that use a component and try to access them; if we succeed, then the component is considered operational, if not, then we have detected a fault. Specifically, if we are testing the postscript renderer, we can select a document and attempt to display it; if we succeed, when using this technique, the system will assume that it will be able to render all documents stored in that format.

- *Partial checks:* This technique probes a small part of the documents. We thus assume that if "part" of the document is not damaged, the whole document is undamaged. For example, if we have a Postscript document, we can just try to print the first page; if we succeed, then we can assume that the rest of the pages can be printed too. Similarly, to test a disk, we only check a few blocks; if those blocks are undamaged, then we assume that the whole disk is undamaged.

- *Predictors:* We can use "indirect" information to predict the likelihood of a fault in a component. For example, if a new format has been released as a replacement of a format in which we have documents stored, we can use this information as a predictor of the obsolescence of the old format. Similarly, some hardware systems provide warnings of the approaching end of the life of the device. For example, Compaq's Intellisafe [16], and Seagate's S.M.A.R.T. [60] measure several disk attributes, including head flying height, to predict failures. The disk drive, upon sensing degradation of an attribute, such as flying height, sends a notice to the host that a failure may occur. Upon receiving this notice, users can take steps to protect their data.

After we have detected that a failure has occurred or if we are predicting an imminent failure, we should repair the component as soon as possible. The longer we wait for the component to be repaired, the higher the risk that another failure will lead to information loss. There are many standard techniques for fast repair, such as having spares available, having parts available and the training of repair personnel.

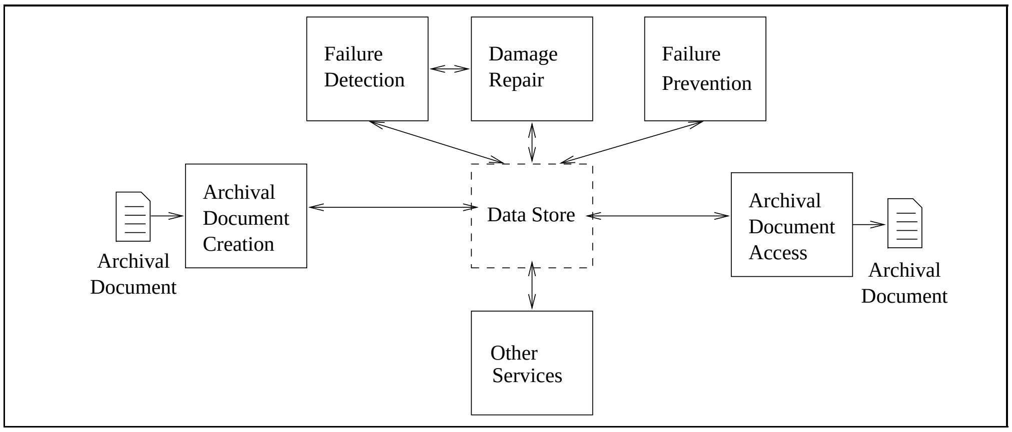

Figure 2.3: Archival Repository Model

2.3 Architecture of an Archival Repository

Our goal in this section is to identify the elements of a typical AR, so we can model each element and study how it impacts reliability. A typical AR stores documents in a *data store* that can fail. The AR can still achieve long-term survivability by enhancing the data store with an *archival system* (AS) that implements some of the techniques of Section 2.2.3. We present our AR model in Figure 2.3. The figure shows the AS modules (in solid-line boxes), the non-fault-tolerant store (in a dashed-line box), and the archival documents. The arrows represent the runtime interactions between the elements.

2.3.1 Archival Documents

In our architecture, an archival document embodies information. An archival document cannot be just a bag of bits, but it must also include components necessary to transform the bits into a human comprehensible form.

An archival document is an abstract entity. The connection between document and access to it is achieved through *materializations.* A materialization is the set of all the components necessary to provide some sort of human access to a document. For example, a materialization may include the bits, disks, and format interpreters

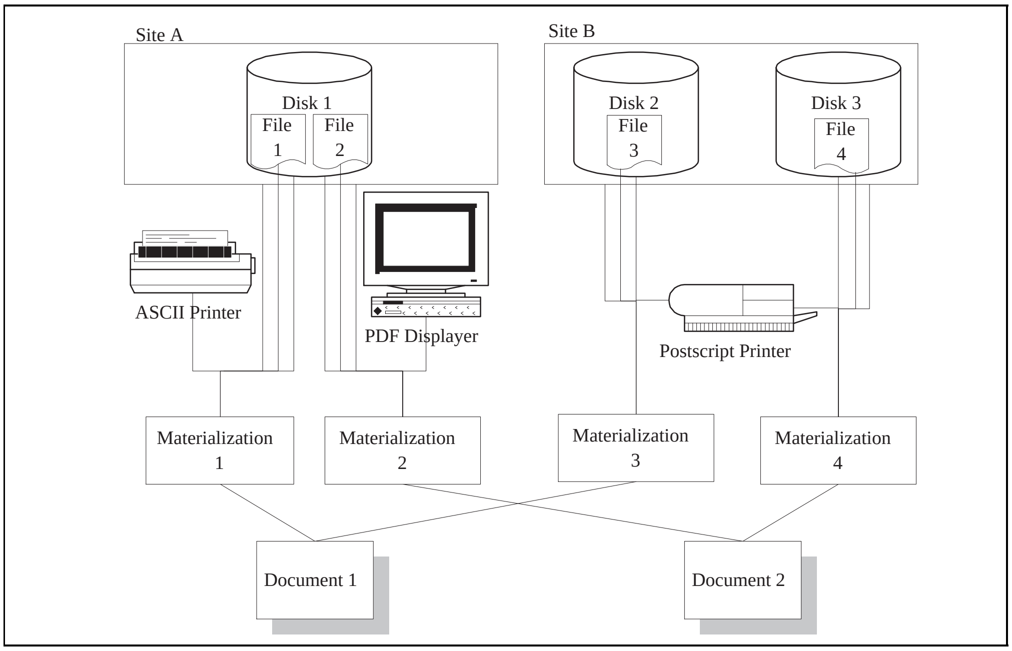

Figure 2.4: Materializations

necessary to display a technical report. The same technical report may be accessible through a different materialization, which may include a different format interpreter to print the technical report.

We illustrate materializations in Figure 2.4. In the figure, there are two archival documents. Each of these documents has two different materializations. For example, Materialization 1 requires the following components to be available: File 1, Site A, Disk 1, and a ASCII printer. Incidentally, note that File 1 is stored on Disk 1, which in turn is at Site A. Such component interdependencies will be discussed later, when we model materialization failures.

The AR is able to preserve documents by preserving the materializations and their components. In this chapter, we treat the documents as "black boxes." We do not attempt to take advantage of document structure (e.g., chapters in a book).

2.3.2 Architecture of the Non-Fault-Tolerance Store

The *Store* encompasses the set of components, such as sites, disks, or format interpreters that make materializations accessible. Because the store is not fault tolerant, materializations may be lost. A materialization is considered lost when *any* of its components has failed. If *all* of the materializations of a document are lost, then the document is considered lost.

To create a materialization, first we must ensure that the necessary components exist in the store. Then we create a record that links together the components as a materialization. For example, say we want to create a document materialization that requires the bits in file "doc.ps," which are located in $disk_2$ in $site_1$ and requires a postscript interpreter. Any components that do not exist already (e.g., the file doc.ps) are created. In some cases, "creating" a component may require a physical action, e.g., adding a new printer or disk to the store. Once the components exist, a metadata record containing references to the components is created (e.g., $\langle site_1, doc.ps, disk_2, postscript \rangle$). This record serves as the identifier for the materialization.

A document metadata record includes the records for all available materializations. If this metadata record is corrupted or lost, the document will not be accessible. Therefore, the record must be one of the required components for any materialization.

2.3.3 Architecture of the Archival System

The AS provides fault tolerance by managing multiple materializations for each document. The AS monitors these materializations, and when a failure is detected, attempts to repair them. The AS can improve fault tolerance further by taking preventive actions to avoid failures. The AS provides the user with the ability to create and retrieve archival documents. It also provides miscellaneous services such as indexing, security, and document retirement, among others. When a user requests a document, the AS uses its metadata to find all the available materializations of that document, selects one and returns it to the user. In this section, we describe the six modules that make up an AS (see Figure 2.3).

The *Archival Document Creation module* (ADC) generates new documents, implementing policies on the number and types of materializations that are needed. For example, say an administrator has decided that documents should be materialized as illustrated by Document 1 in Figure 2.4. Then, for each new document, the ADC creates (on the data store) the appropriate "File 1" and "File 2," the document metadata record, and checks that the other components exist. The main objective of this chapter is to provide a framework that allows a system administrator to choose the best materialization policies to achieve a desired level of reliability.

The *Archival Document Access module* (ADA) services requests for documents. Basically, the module translates the request for a document into a request for the appropriate components of one of the document materializations.

The *Failure detection module* (FD) scans the store looking for damaged or lost materializations. When a damaged materialization is found, the failure detection module informs the Damage Repair module (described below) about the problem.

The *Damage Repair module* (DR) attempts to repair damaged documents. There are many strategies to repair a damaged document, as discussed in Section 2.2.3. The input of the DR module is a signal from the FD module.

The *Failure Prevention module* (FP) scans the store and takes preventive measures so materializations are less likely to be damaged. For example, the FP module may copy components that are stored on a disk that is close to the end of its expected life, into a newer disk.

Finally, the *Other Services module* (OS) provides miscellaneous services such as indexing, security, and document retirement. Retiring a document involves removing from the store any components that are no longer needed, even by other materializations.

2.4 Failure and Recovery Modeling

In this section, we will model the failure and recovery characteristics of an AR, based on the architecture presented in the previous section. First, we will explore how to model a non-fault-tolerant store, and then the archival system (AS). Later, we will combine these two models into an archival document model.

2.4.1 Modeling a Non-Fault-Tolerance Store

To model the failure characteristics of a store, we start with an abstract representation of materializations and components. We model a materialization as an n-tuple $\langle mat_{id}$, doc_{id}, $comp_1$, $...comp_n\rangle$; where mat_{id} is the materialization identifier, doc_{id} is the document identifier, and $comp_1...comp_n$ are the components required to provide the required document access. The identifiers mat_{id} and doc_{id} together form a unique id for the materialization. For example: $\langle M1$, $TR1233$, *doc.ps*, $site_1$, $disk_2$, $postscript\rangle$ means that the materialization $M1$ contains the document identified by $TR1233$ that needs the bits in file *doc.ps*, $disk_2$, $site_1$ and the postscript interpreter in order to be readable. A document can have more than one materialization. For example, Technical Report 1233 can also have the materialization $\langle M2$, $TR1233$, *doc.ps*, $site_1$, $disk_3$, $postscript\rangle$, which would be a copy of M_1 but on a different disk ($disk_3$).

We model components by a tuple $\langle component_{id}$, $type\rangle$, where $component_{id}$ is a unique identifier for the component instance and *type* is the class (e.g., file, disk, interpreter) to which the instance belongs.

To further model components, we need to describe:

- How many component instances and types are present in the system: that is, how many disks, formats, etc., are available.

- Failure distribution of each component type. Many components have two different failure distributions, one during archival and another during access. For example, a tape is more likely to fail when it is being manipulated and mounted on a reader than when it is stored. Therefore, each component will have two failure distributions: during archival (i.e., time to next failure when the component is not used) and during access. For some components, such as disks or sites, the access and archival distributions will be the same; but for other components, such as tapes or diskettes, these distributions can be very different.

- Time distribution for performing a component check. This distribution describes how long it takes to discover a failure (or to determine that a component is good), from the time the check process starts. For example, consider checking a tape. This may involve getting the tape from the shelf, mounting the tape, and scanning the tape for errors.

- Time distribution for repairing a component failure. This distribution describes how long it takes to repair a component. This distribution may be deterministic (if the component can be repaired in a fixed amount of time). Repair time may be "infinite" if the component cannot be fixed.

In addition, there is an important interdependency between components. Specifically, the failure of one component may cause the failure of another component. For example, if a site fails (e.g., because it was destroyed by a fire), then all the disks at the site will also fail. As pointed out earlier, we are only taking into account permanent failures; transient failures (e.g., the site was temporarily disconnected from the network) are ignored. This failure dependency is captured by a directed graph. For example, an arrow between "Site A" and "Disk 1" in the interdependency graph means that if "Site A" fails, then "Disk 1" will also fail.

We close this subsection with two comments. First, we do not claim that the model presented for the store is complete. For instance, we have not included policies for handling concurrent access. There is always a tradeoff between complexity of the model and our ability to analyze it. We believe that our model strikes a good balance in this respect, and captures the essential features of a store. Second, the reliability predictions we make are only valid for the *current* configuration of the repository. Over time, the repository will change (e.g., as new devices are introduced), so we may need to change our repository model. As the model changes, we may need to revisit our predictions.

2.4.2 Modeling an Archival System

In this section, we describe how to model the behavior of the modules of the archival system. We do not include failure distributions for these modules as we are assuming that the AS itself does not fail. We recognize that this is a strong assumption, but in this chapter, we have chosen to concentrate on the failure of components, instead of on the failure of the system that provides fault-tolerance. How to develop error-tolerant robust software design has been studied in [63].

In general, we model the input of the modules with probability distribution functions and their behavior by algorithms. For example, consider the document creation (ADC) module. Its input distributions tell us how frequently requests for document creation arrive, how many materializations each new document will have, and which components will be selected to participate in a given materialization. The algorithms for the failure detection (FD) module spell out what policies are implemented, e.g., if all components are checked on a regular basis or not.

The probability distributions that drive the model can be obtained in different ways. If we have data from a real system, we can use the data directly (trace driven), or we can define an empirical distribution, or we can fit the data onto a theoretical distribution [4]. If we do not have real data, we need to choose a theoretical distribution that matches our intuition. A sensible distribution to choose (when requests are generated independently) is a Poisson distribution for event inter-arrival times [12].

We summarize the model parameters in Figure 2.5. The figure is divided in three parts. At the top are the parameters that describe the AR: the number of components and their types, and the failure dependency graph. Then, we list all the distributions needed for the model with the units being modeled in parenthesis. Finally, we list all the policies and algorithms that must be defined to model the archival system.

2.4.3 Modeling Archival Documents

In this section, we combine the models for the data store and the AS, in order to describe the life of an archival document. In Figure 2.6, we depict our model for the life of an archival document. The life of a document starts when its materializations are created by the ADC module and handed to the store. The creation of a document may not be an instantaneous process. For example, if long-term survival is achieved by keeping multiple copies, the document is not considered archived until all the

- **AR Description**
  - Number of components and types
  - Failure dependency graph
- **Distributions**
  - For each component type:
    - Failure distribution during access (time)
    - Failure distribution during archival (time)
    - Failure detection distribution (time)
    - Repair distribution (time)
  - Document creation distribution (time)
  - Document access distribution (time)
  - Access duration distribution (time)
  - Document selection distribution (document)
- **Policies**
  - Document Creation policy
  - Document to materializations policy
  - Failure detection algorithm
  - Damage Repair algorithm
  - Failure prevention algorithm

Figure 2.5: Archival Repository Model Parameters

copies are generated. Once the ADC module has taken all the actions that ensure the long-term survival of the document, then the document has full protection, and we say that the document is in the *Archived* state.

When any of the document components fail (based on the distributions in Figure 2.5), the document is considered to be in the *Damaged* state and becomes temporarily unprotected. For example, if we keep two copies of a document and one of the copies is lost, then the document would be damaged. As explained earlier, the AS will not know that a document is damaged until the FD module detects the failure. When the failure is detected, the document goes to the *Damage Detected* state.

When damage to a document is detected (by the policies summarized in Figure 2.5), the AS starts actions to restore the document and, hopefully, return it to the Archived state. For example, if the document is damaged because one of its copies

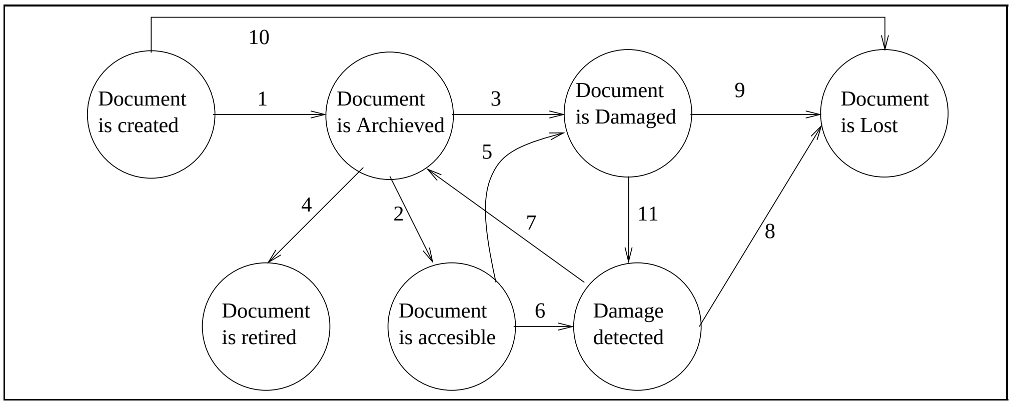

Figure 2.6: Archival Document Model

was lost, the repository can replace the damaged copy by creating a fresh one from one of the good copies. However, if the repair is not successful, then the document may be *Lost*. This latter state is the one that we want to avoid in an archival system.

We also distinguish two additional states: Accessible and Retired. When a document is in the *Accessible* state, it can be accessed (e.g., read, printed) by users, which is not the case for the Archived state. For example, if some of the document's components are stored on a tape which is kept in a safe, we need to take the tape out and mount it in a reading device to make it accessible. When the tape is stored, the document is in the Archived state; when the tape is mounted, it is in the Accessible state. As we discussed in the previous section, by making the document accessible, in general, we are increasing the chances of damaging the document. Therefore,the probability of transition 7 is, in general, greater than the probability of transition 5.

The Retired state allows users to mark the documents that are not needed anymore. In this case, the document is retired from the archival system and the system does not provide any survivability guarantees. It is important to note that retiring a document may eventually result in removing all materializations from the store. This action is different from taking a document "out of circulation," in which case the document is not longer available to regular users, but it is still preserved for historical reasons.

In the Archival Document Model, transitions between states are due to deliberate actions taken by the AR or due to external events. An example of a deliberate action is the creation of enough copies that lead to the transition between the Created state and the Archived state. An example of an external event is the failure of a disk, causing the transition of a document from Archived to Damaged. Most external events have a probabilistic nature.

The archival document model presented contains only the *main* states for a document. In some particular scenarios some of the states may merge or may not exist at all. For example, in an on-line archival system, there may not be any difference between the "Archived" and "Accessible" states. Similarly, depending on the actual policies of the AR, some of the states can be expanded. For example, if a system has a "Document to Materialization" policy that requires the creation of three copies of the document, the Damaged state expands into six states, one for each possible combination of one or two copies being lost. It is important to note that for analyzing an archival system we do not need to expand the nodes unless we are interested in the specific probability of being in those expanded nodes.

Also note that we have simplified our document model by assuming some transitions do not occur. For example, we do not show a transition directly from the Archived state to the Lost state, as we assume that documents are always damaged before they are lost. We also do not show a transition between the Damaged state and the Archived state, as we assume that damage does not repair itself.

2.4.4 Example: An Archival System based on Tape Backups

Let us describe the archival document model in more detail by using a specific example (Figure 2.8). This archival system saves documents on two tapes. The tapes are stored in a controlled environment (i.e., a safe) to improve preservation. We summarize the parameters for the model of this system in Figure 2.7.

The operation of the system starts with the ADC saving n documents on each of the tapes. For simplicity in this example, we will assume that this operation always succeeds. Let us assume that the time between access requests to the ADA module is

- **AR Description**
  - Initial collection: empty.
  - Number of components and types: 2 tape drives
  - Failure dependency graph: empty
- **Distributions**
  - Tape Failure distribution during access: exponential with rate D
  - Tape Failure distribution during archival: exponential with rate L
  - Tape Failure Detection distribution : exponential with rate R
  - Tape Repair distribution: uniform with probability 1.
  - Document creation rate: n documents at startup, then no documents are created.
  - Document access period: A
  - Access duration period: S
  - Document selection: uniform over the n documents.
- **Policies**
  - Document Creation policy: write in both tapes.
  - Document to Materialization: read from any working tape.
  - Failure detection algorithm: complete scan of both tapes taking a negligible amount of time.
  - Damage Repair algorithm: discard bad tape and replace with new copy taking a negligible amount of time.
  - Failure prevention algorithm: none

Figure 2.7: A model for a dual tape system

distributed exponential with rate A. The duration of a successful tape access is also exponential with a rate $1-S$. The FD module scans the tapes and finds all failures at a rate R. The DR module can repair all failures as long as one of the tapes is working and the repair is done instantaneously. Therefore, the system will lose the documents if both tapes fail in between the scans of the FD module. The system does not have an FP nor an OS module.

Additionally, let us assume that while the tapes are in a safe environment, they become unreadable under an exponential distribution with rate L. When the tapes are being read, the rate of damage increases to D ($D > L$).

Using the information from the archival model, we can model the life of a document

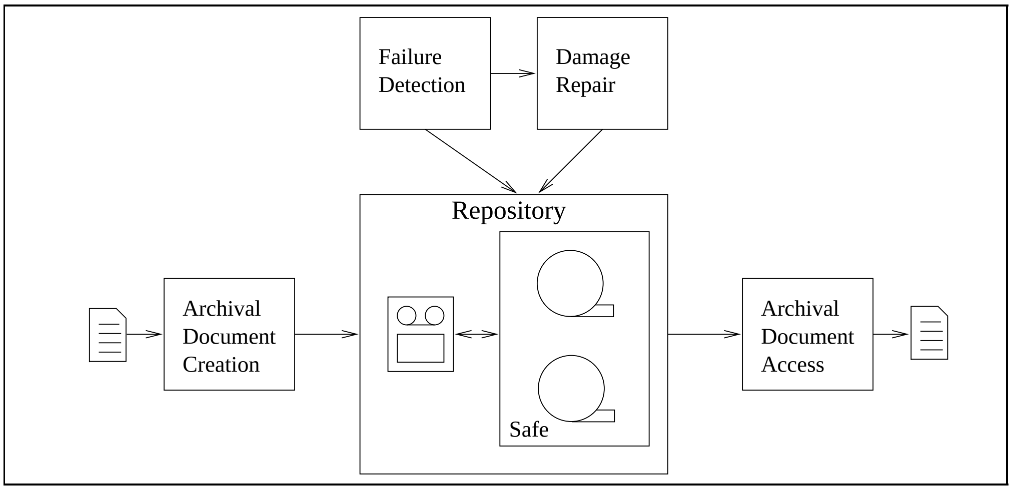

Figure 2.8: A Dual-tape Archival System

as it is presented in Figure 2.9. The document is considered Archived after we finish copying the document onto the tapes and the tapes are in a safe place. As this process always succeeds, the transition to Archived happens with probability 1.

Given that we have two tapes, then the distribution for transition from Archived to Damaged will be exponential with rate $2L$ (if we assume independent failures). When making the document accessible, we increase the rate of failure of a tape to D, thus, the rate of the transition from Accessible to Damaged will be $D + L$, D for the mounted tape and L for the tape that stays in storage. Finally, when a tape is damaged, the system will attempt to recover the tape by using the other tape. We are assuming that this recovery will succeed with probability R, in which case we will return to the Archived state. However, if while in a Damaged state, the second tape becomes unreadable (this will happen with probability L), then the document would be lost. The rate of access to the document is A (not A/n, where n is the number of documents); this is because when accessing a document, we are actually accessing all documents on the tape.

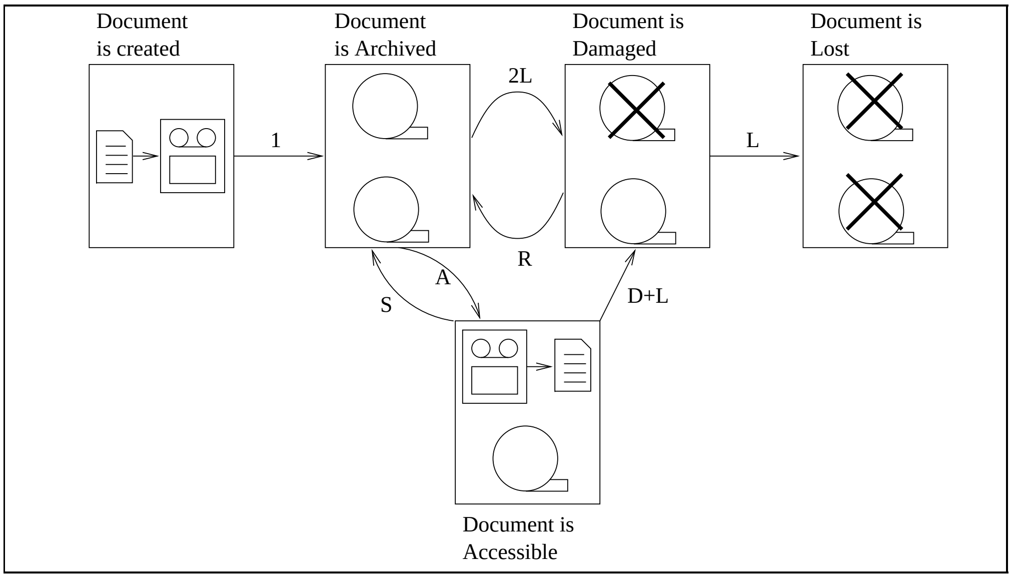

Figure 2.9: A Dual-tape Archival System

Analyzing the Dual-tape Archival System

Our goal is to find how long it would take for the Dual-tape Archival System to lose documents. In other words, we want to find the mean time to failure (MTTF) of the system. The computation of the MTTF of an AR can be done in two ways: we can attempt to compute it analytically, or we can infer it statistically by simulating the system. The analytical approach consist in modeling the problem as a Markov Decision Process (MDP) and finding a solution for it. However, MDPs make strong assumptions that may not be true in archival systems and the computational complexity of finding a solution for them is high. The basic assumption of Markov models is that the probability of a given state transition depends only on the current state [74]. Moreover, the amount of time already spent in a state does not influence either the probability distribution of the next state or the probability distribution of remaining time in the same state before the next transition. This assumption prevents us from using realistic failure distributions. In addition, in [62] the computational complexity of solving MDPs has proven to be P-complete. This means that, although the problem is solvable in polynomial time, if an efficient parallel algorithm were available, then all problems in P would be solvable efficiently in parallel (which is considered an unlikely outcome at the moment). In practice, this result means that given the high order of the polynomials describing the complexity of solving a MDP, it would take months to solve a *general* MDP with a million node/transitions. In the next section, to overcome the limitations of solving MDPs, we will introduce a Monte-Carlo simulation tool that allows the use of general failure distributions and is able to find the MTTF of complex archival systems in a short time.

The Dual-tape Archival System is simple enough that it can be modeled using a Markov Chain and the Mean-Time to failure can be computed exactly:

$$MTTF = \frac{S \cdot R + D \cdot R + D \cdot L + 2L^2 + 4L \cdot R + 3L \cdot S + 2A \cdot L + 2A \cdot R}{2L^3 + 2L^2 D + A \cdot L^2 + A \cdot D \cdot L + 2L^2 S} \tag{2.1}$$

Formal modeling allows us to take precise allocation decisions. For example, if we use the parameters in Figure 2.10, then the MTTF of the system will be 9,316 days (or about 25 years). We can also use this result to make resource allocation decisions. Let us suppose that we want to improve the MTTF to 100 years and we can do so by improving the rate of detection and repair of damaged media (i.e., change the value of R). In this case, we can compute the needed repair frequency to achieve a MTTF of 100 years: 13 days. These 13 days include the sum of the frequency of checking the tapes plus the number of days that it takes to replace a defective tape with a new copy. Specifically, if it takes 1 day to create a new tape, then we need to check all the tapes every 12 days.

The computation of Equation 2.1 is complex, even though we are not considering software errors, sites failure, etc. When considering all those factors, obtaining a closed result is very difficult, and in some cases, impossible. In the next section we will introduce a simulation tool, ArchSim, that will be able to handle more complex cases.

Mean-Time to failure (days)	
Parameter	*value*
Tape in stable storage ($1/L$)	1000
Tape when reading ($1/D$)	250

Access Rates (days)	
Parameter	*value*
Access frequency ($1/A$)	20
Access duration ($1/S$)	2

Failure detection and Repair Rate (avg. days)	
Parameter	*value*
Tape ($1/R$)	60

Figure 2.10: Parameter values

2.5 ArchSim: A Simulation Tool for Archival Repositories

To evaluate a possible AR configuration, we need to predict how well it protects documents. This prediction can sometimes be done analytically, but as the AR gets more complex, an analytical solution is impractical (and sometimes impossible). Instead, we rely on a specialized simulation engine for archival repositories: ArchSim. We start this section by discussing the specific challenges confronted when simulating an AR. Then we describe ArchSim and its libraries.

2.5.1 Challenges in Simulating an Archival Repository

ArchSim builds upon existing simulation techniques for fault-tolerant systems. However, the unique characteristics of archival repositories make their simulation challenging:

- *Time Span:* The life of an archival system is measured in hundreds, perhaps thousands of years. This means that simulation runs will be extremely long, so special precautions must be taken to make the simulation very efficient. Furthermore, given these long periods, failure distributions must take into account component "wear-out." (A component is more likely to fail after 50 years than when it is new.) Simple failure distributions (e.g., exponentially distributed time between failures) are frequently used in fault-tolerant studies, but they cannot be used here since they do not capture wear-out.

- *Repairs:* In an archival system we cannot in general assume that damaged components can always be replaced by new identical components (another common assumption when studying fault-tolerant systems). For example, after 100 years, it may be impossible or undesirable to replace a disk with one having the same failure characteristics.

- *Component models:* Component models are fairly rich, compounding the number of states that must be considered. For instance, as we have discussed, a file is not simply correct or corrupted. Instead, it can be corrupted but the error undetected, it can be correct but not accessible for reads, and so on. The failure models in each of these states may be different; e.g., a file is more likely to be lost when being read.

- *Sources of failures:* A document can be lost for many reasons; e.g., a disk fails or a format becomes obsolete. Each of these failures has very different models and probability distributions. The approach of finding the "weak link" and assuming that all other factors can be ignored is not appropriate for ARs.

- *Number of Components:* An AR needs to deal with a large number of components and materializations. The challenge of simulating a large number of objects has been studied extensively [29, 59] and ArchSim uses those results.

2.5.2 The Simulation Engine: ArchSim

ArchSim receives as input an AR model, a stop condition (stop when the first document is lost or when all documents are lost) and a simulation time unit (minutes, hours, days, etc.). ArchSim outputs the mean time to failure (mean time to stop condition), plus a confidence interval for this time. We are currently considering other output metrics, e.g., the fraction of the documents that are available after some fixed amount of time. However, these other metrics are not used in our case study (Section 2.6).

To define the AR model, each distribution and policy is implemented as a Java object, so they can be as general as necessary. For example, for component repair, the corresponding Java module can simply use a probability distribution (perhaps one of the library functions described below) to generate the expected repair time. However, that module can easily be replaced by one that first decides if the component can be repaired (using one probability distribution), and then for each case generates a completion time (when the component is repaired or the repair is declared unsuccessful).

2.5.3 Library of Failure Distributions

ArchSim makes available a library of pre-defined failure distributions that can be used to describe AR components. The distributions in the library are: deterministic, uniform, exponential, infant mortality, bathtub, and historical survival. To illustrate, in Figure 2.11(a)-(e), we sketch generic versions of these probability distributions. (The area under these curves, from time 0 to t, represents the probability the component will fail by time t.)

1. Deterministic Model (graph not shown): This is a deterministic model in which the event happens (with probability one) at a fixed time. It is useful for describing deterministic events such as the start of a failure detection process. The mass function of this distribution is:

$$p(t) = \begin{cases} 1 & \text{if } t = t_0; \\ 0 & \text{otherwise.} \end{cases}$$

   where t_0 is the time when the event will happen.

2. Uniform Model (Figure 2.11a): This is a very simple model based on a uniform distribution. Uniform distributions are convenient, as they are easy to generate and analyze. The density function of a uniform distribution is:

$$f(t) = \begin{cases} \frac{1}{t_1 - t_0} & \text{if } t \in [t_0, t_1]; \\ 0 & \text{otherwise.} \end{cases}$$

where $[t_0, t_1]$ is the time interval where the failure may happen.

3. Exponential model (Figure 2.11b): This model assumes that the probability of failure at any single point in time is constant (i.e., the model is memoryless). In many fault-tolerance systems, exponential distributions are used instead of more complex distributions (described next) under the assumption that components are checked before installation and that the life of the system is much shorter than the life of the components. However, in an archival system, these assumptions are not always true. The density function of the exponential distribution is:

$$f(t) = \begin{cases} \frac{1}{\beta} e^{-t/\beta} & \text{if } t \geq 0; \\ 0 & \text{otherwise.} \end{cases}$$

where β is the mean (expected time to failure) of the distribution.

4. Infant Mortality model (Figure 2.11c): This model is good for describing failures due to system or human errors. At first, when the document has been recently created, the probability is high (as there could be errors in its creation and/or initial storage); then the probability drops sharply and remains fairly constant during the rest of the life of the document. The density function of this distribution can be obtained by using a Weibull distribution in the infant mortality phase and an exponential distribution in the constant phase:

$$f(t) = \begin{cases} \alpha \beta^{-\alpha} t^{\alpha - 1} e^{(-t/\beta)^{\alpha}} & \text{if } t \in [0, t_1); \\ \frac{1}{(\beta/t_1)^{\alpha} t_1} e^{-(t-t_1)/(\beta/t_1)^{\alpha} t_1} & \text{if } t \in [t_1, \infty); \\ 0 & \text{otherwise.} \end{cases}$$

where $[0, t_1)$ is the infant mortality period, $[t_1, \infty)$ is the constant period, α $(0 < \alpha < 1)$ is the shape parameter (values closer to zero generate higher rates of early failures), and β $(\beta > 0)$ is a scale parameter used to adjust the mean of the distribution during the infant mortality phase.

5. Bathtub model (Figure 2.11d): This probability failure function is considered typical of electronic devices. In the bathtub distribution, the probability of failure early in the component's life (infant mortality phase) and late in its life (wear-out phase) are higher than during the middle years. The density function of this distribution can be obtained by using a Weibull distribution in the infant mortality phase, an exponential distribution in the constant phase, and another Weibull distribution in the wear-out phase:

$$f(t) = \begin{cases} \alpha\beta^{-\alpha}t^{\alpha-1}e^{(-t/\beta)^{\alpha}} & \text{if } t \in [0, t_1); \\ \frac{1}{\gamma}e^{-(t-t_1)/\gamma} & \text{if } t \in [t_1, t_2]; \\ \delta\zeta^{-\delta}(t-t_2)^{\delta-1}e^{(-(t-t_2)/\zeta)^{\delta}} & \text{if } t \in (t_2, \infty); \\ 0 & \text{otherwise.} \end{cases}$$

where $[0, t_1)$ is the infant mortality period, $[t_1, t_2]$ is the constant phase, (t_2, ∞) is the wear-out phase, α $(0 < \alpha < 1)$ is the shape parameter for the infant mortality phase, β $(\beta > 0)$ is a scale parameter used to adjust the mean of the distribution during the infant mortality phase, γ is the mean of the constant phase, δ $(\delta \geq 1)$ is the shape parameter for the wear-out phase, ζ $(\zeta > 0)$ is a scale parameter used to adjust the mean of the distribution during the wear-out phase. The constants in the distribution must be chosen such as: $e^{-(t_1/\beta)^{\alpha}} + e^{-t_2/\gamma} = e^{-t_1/\gamma} + e^{-(t_2/\zeta)^{\delta}}$

6. Historical survival model (Figure 2.11e): This model attempts to describe the life of a published resource. The model is based on the idea that if a resource survives after a long period, it becomes of "historical significance," and its chance of continual survival increases. Although there have not been studies modeling the obsolescence of formats, we theorize that this function is a good

choice for modeling the obsolescence of a format interpreter. When a format is adopted and retained for a long time (we are assuming here that we are adopting a "commercial" format), the probability of not being able to interpret the format is low. As the format ages, the probability of failure increases as knowledge about the format starts disappearing until we reach the historical phase when the probability of losing the format, after having survived that far, starts decreasing. This distribution is modeled using a Weibull distribution for the adoption phase and another Weibull distribution for the historical phase.

$$f(t) = \begin{cases} \alpha\beta^{-\alpha}t^{\alpha-1}e^{(-t/\beta)^{\alpha}} & \text{if } t \in [0, t_1); \\ \delta\zeta^{-\delta}(t-t_2)^{\delta-1}e^{(-(t-t_2)/\zeta)^{\delta}} & \text{if } t \in (t_1, \infty); \\ 0 & \text{otherwise.} \end{cases}$$

where $[0, t_1)$ is the adoption phase, (t_2, ∞) is the historical phase, $\alpha \geq 1$ is the shape parameter for the adoption phase (the higher the parameter, the lower the probability of losing the format), $\beta > 0$ is a scale parameter used to adjust the mean of the distribution during the adoption phase, $(0 < \delta < 1)$ is the shape parameter for the historical phase (the closest to zero, the "earlier" a failure may happen), ζ $(\zeta > 0)$ is a scale parameter used to adjust the mean of the distribution during the historical phase. The parameters in the distribution must be chosen such as: $(t_1/\beta)^{\alpha} = (t_2/\zeta)^{\delta}$

2.5.4 ArchSim's Implementation

ArchSim follows the structure of a traditional simulation tool. Each module of the AR model can register future events in a timeline. For example, when a disk is created, the simulation uses the disk failure distribution to compute when the disk will fail; then, it registers the future failure event in the timeline. The simulation engine advances time by calling the module that registered the first event. This module may change the state of the repository and register more events in the timeline. After the module returns, the simulation advances to the next event, in chronological order. The user

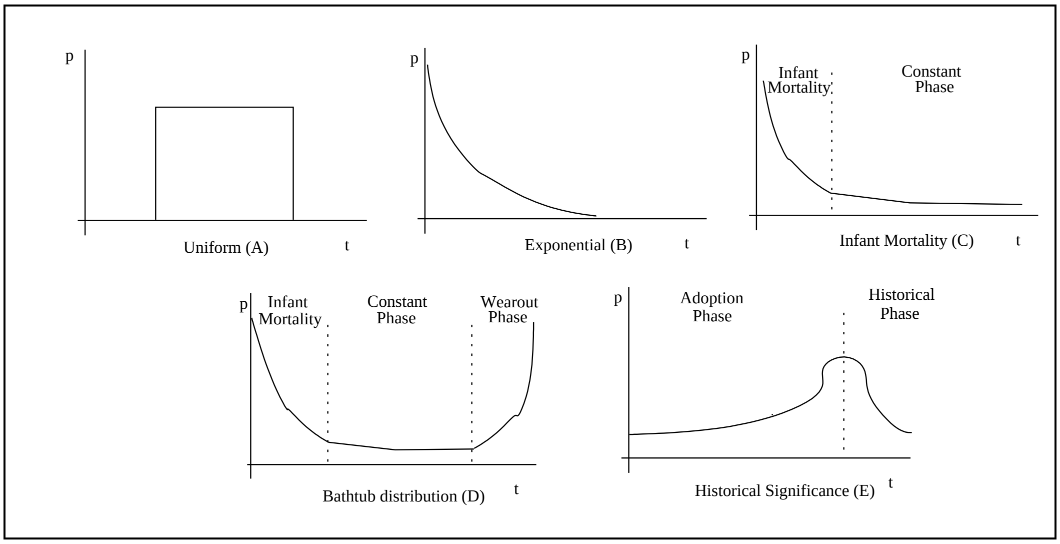

Figure 2.11: Failure Functions

can choose between two end conditions for the simulation: the simulation can stop after the first document is lost or after all the documents are lost.

ArchSim needs to be very flexible and efficient to meet the challenges of simulating an archival repository. Flexibility is needed to model very different archival conditions and implementations. Speed is needed to cope with many materializations, components, and events. Additionally, each simulation needs to be run many times in order to obtain narrow confidence intervals.

In an AR simulation, many events are inconsequential. For example, suppose that the detection module schedules periodic detection events. If the detection event finds a fault (i.e., there was a failure event before the detection event), then the module starts a component repair; if no failure is detected, then the module does nothing. If a repair module checks a component with MTTF of 20 years every 15 days, on average, 486 events ($20 * 365/15$) will be fired and the repair module will return without doing anything; only in event 487, when a failure of the component has occurred, will the repair module perform an action. Given the large number of components and modules that may be part of the model, this large number of inconsequential events represents a significant overhead. To avoid this overhead and to improve efficiency, modules are allowed to register *conditional* events in the timeline. These events will only happen if some other event happens before them. By using conditional events, we can condition the firing of the detection event only if a failure event on a specific component happens before it. A conditional event is not registered directly in the timeline. Instead, it is registered in an index that is part of its triggering event. For example, if event B is conditional to the occurrence of event A, we will put B in the index of A. When a new event A is scheduled in the timeline, we look in the index and find that B is conditional to it, so at that point we also schedule B.

To reduce the number of events further, ArchSim also modifies failure distributions. For example, when modeling preventive maintenance, a large number of "replace component" events are generated. We can eliminate all these events, by modifying the failure distribution. Specifically, the original failure distribution is used to generate the time of the next failure of the component. If this time is higher than the prescribed preventive maintenance period, we ignore this time, and we generate a new failure time, again using the original distribution. We repeat this process until the failure time is lower than the preventive maintenance period. The new distribution then returns the number of iterations minus one, times the PM period, plus the last failure time.

Another challenge for ArchSim is how to deal with a large number of materializations and components. This is done by scheduling only the *next* failure for each component type and associating a trigger with that event. When the failure event happens, the trigger is activated. The trigger computes when the *next* failure of a member of that component type will occur, and adds it to the timeline. Obviously, this approach is only beneficial if we have many components of the same type, which is a reasonable scenario for an AR. In the case in which we have N different distributions for N components, this technique does not improve the simulation time, but it does not increase it.

2.6 Case Study: MIT/Stanford TR Repository

In this section, we use ArchSim to answer some design questions for a hypothetical MIT/Stanford Technical Report Archival Repository. The AR follows the Stanford Archival Vault (SAV) design and implementation [17]. In this case study, MIT and Stanford preserve their Computer Science Technical reports by replicating the reports at both universities. In this case study we will have to make many assumptions. Our goal here is *not* to make any specific predictions, but rather to illustrate the types of evaluations that ArchSim can support, the types of decisions that must be made to model an AR, and the types of comparisons than can be made to support rational decisions among alternatives. In addition to this case study, we also used ArchSim to evaluate the dual tape system presented in Section 2.4.4. The results from ArchSim and the Markov chain analysis were consistent for the parameter values of Figure 2.10.

We assume that the collection has 200,000 documents and that each document is stored in one or more of four available formats. The repository has two types of components: storage devices (disks) and format interpreters. To ensure preservation, the AR maintains four materializations of each technical report; two materializations at Stanford and two at MIT. Materializations are created by choosing two formats out of the four available formats. Two of the materializations will be in one of the chosen format, while the other two will be in the other. Then, at each site, we place two materializations that are in different formats on two different disks. Figure 2.12 illustrates the arrangement of the technical reports in this system.

Disks, formats, and sites have uniform failure distributions with parameters $1/\phi_{sto}$, $1/\phi_{form}$, $1/\phi_{site}$. As our base values, we are assuming ϕ_{sto}, ϕ_{form}, and ϕ_{site}, the mean time to failure (MTTF) for disks, formats and sites, to be 3, 20, and 45 years respectively.

These values are our best guess for a typical archival system. We chose 3 years as the disk MTTF, as this is the normal period under which a hard drive is under manufacturer guarantee. We chose 20 years for format MTTF as we theorize that it will take that long for a well-known format to be replaced by a new format and for all displayers and transformers of the old format to be lost. We chose 45 years for

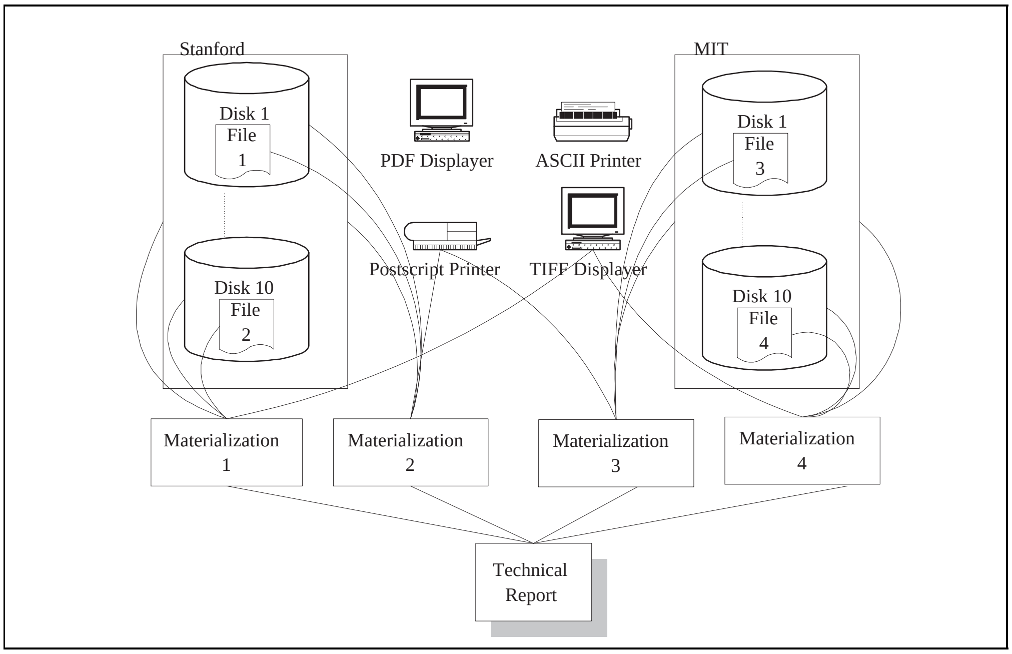

Figure 2.12: MIT/Stanford CSTR Scenario

the site MTTF as we assign a 50% probability to the event of losing the Stanford site due to a high intensity earthquake in the San Francisco Bay Area (which is predicted to happen within a 45-year period).

The archival system checks for faults periodically. When a fault is detected in one of the storage devices, the bad device is retired, and a new device is set online. Then, the system regenerates the bad device by making a copy of the lost materializations from the other site. Similarly, when a format becomes obsolete, a new format is selected and a new set of materializations (transformed from a non-obsolete format) is created in the new format. In addition, in case of site failure, the site is recreated from the other site. If all sites, formats and devices that support all the materializations of a technical report are lost, then the technical report is lost and the simulation stops. Disks, formats, and sites are checked and repaired (if needed) every ρ_{sto}, ρ_{form}, and ρ_{site} days. As our base values, we are assuming ρ_{sto}, ρ_{form}, and ρ_{site} to be 60, 60, and 7 days.

Parameter	*Symbol*	*value*
Number of disks	n_{sto}	100 per site
Number of formats	n_{form}	4
Number of documents	num_{doc}	200,000
Mean Time to Disk Failure	ϕ_{sto}	3 years
Mean Time to Format Failure	ϕ_{form}	20 years
Mean Time to Site Failure	ϕ_{site}	45 years
Disk Failure Detection/Repair time	ρ_{sto}	60 days
Format Failure Detection/Repair time	ρ_{form}	60 days
Site Failure Detection/Repair time	ρ_{site}	7 days

Figure 2.13: Base values

To justify these values, we need to describe what is involved in the detection and repair of component failures. Detecting a failure in a disk involves scanning the whole disk and checking for lost data. When we find lost data, we need to order a new disk and then copy all the data that was in the damaged disk from other sources onto the new disk. Assuming that the repair time is 60 days means that we need to dedicate only 3% of the disk bandwidth to scan all materializations in order to detect failures. We did not chose a quicker repair because the scanning overhead would be too high in our opinion. For example, 25% of the disk bandwidth is required to detect failures in 7 days.

For formats, the detection/repair times imply that we are able to realize that a format is obsolete and that we can create a new copy of the document from a non-obsolete format within a 60 day period. In the case of site failures, we assume that the detection/repair time for the site is much lower than for formats and disks. The detection itself should be rather fast in this case (the entire site is down), and the 7 days could be the time it takes to find a backup site to take over.

In this case study, we are assuming that failures are total. This is, we cannot partially repair a component and salvage some of the materializations. The failure distribution during access will be assumed to be the same as the failure distribution during archival. The simulation parameters are summarized in Appendix A and the base values for our simulation are in Figure 2.13.

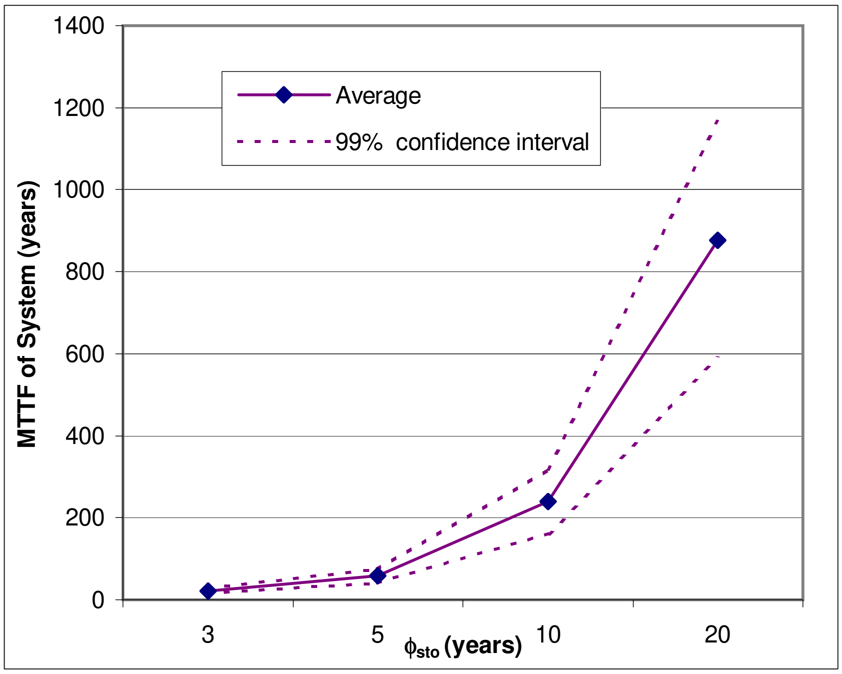

Figure 2.14: MTTF vs. ϕ_{sto} with ρ_{sto} of 60 days

In our first experiment, we evaluate the effect of the failure MTTF and repair times of storage devices on the system MTTF. In Figure 2.14, we show the system MTTF for different disk MTTFs, given a detection/repair time of 60 days. To single out the influence of storage device failures, we are assuming in this experiment that formats and sites never fail. The dotted lines in the figure represent the 99% confidence interval for the simulation, while the solid line is the average of all the simulation runs. As expected, the system MTTF increases when the disk MTTF increases (when the disk failure rate decreases). The exponential shape of the curve is the result of constant repair time. As we keep increasing the disk MTTF, it is much more improbable that another device will also fail before 60 days have passed. This graph allows us to select a good storage device for a target system MTTF. If the library targets a 10-year MTTF, then a disk with a failure MTTF of 3 years will suffice. However, if the library requires a MTTF of 100 years, then we will need disks with a MTTF of about 6 years. Most manufacturers guarantee their disks for three years and very few guarantee them beyond five years. Therefore, a 6-year disk MTTF requirement will be hard (or very expensive) to meet. Nevertheless, we can still achieve our target system MTTF by changing other parameters in our system, as we will see in the next experiment.

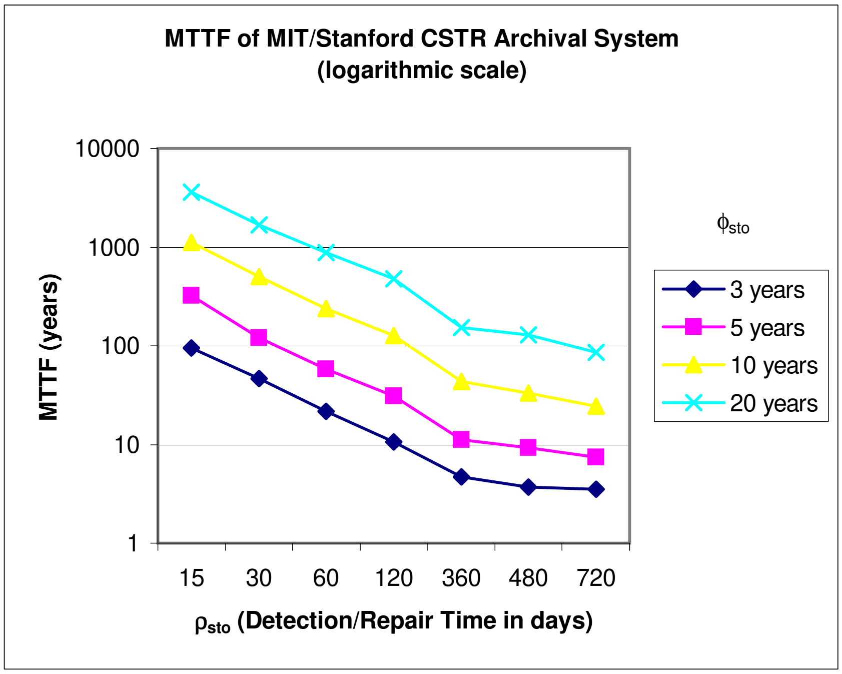

Figure 2.15: System MTTF (logarithmic scale)

We now evaluate the sensitivity of the system MTTF to the repair time (ρ_{sto}). In Figure 2.15, we show the MTTF of the AR for different disk MTTF (ϕ_{sto}) values, and for different expected detection/repair times. (The graph has a logarithmic scale. Confidence intervals are not shown to avoid clutter.) First, let us concentrate on the curve for a ϕ_{sto} of 3 years. As expected, the MTTF of the system decreases when the ρ_{sto} of the storage devices increases. However, the shape of the curve is more interesting. At low detection and repair times, there is a high positive impact on the MTTF of the system. However, as we increase the repair times, the system MTTF drops sharply. Interestingly, for a repair time greater than 120 days, about 1/9 of the

MTTF of the storage device, the effect on the system MTTF of the detection/repair module is small. Thus, the detection and repair times must be much lower than the storage device failure MTTF to have a significant effect on the MTTF of the Archival System.

What is the optimal solution for a given MTTF with respect to disk MTTF repair times? The answer depends on the cost assigned to those two factors. By looking at all the curves of Figure 2.15, we can observe that we can achieve similar MTTF by using better media or by reducing the detection/repair times. For instance, a system that uses a storage device with ϕ_{sto} of 5 years (i.e., a low quality storage device) and has a detection/repair time of 30 days, is as good as a system that uses a high quality storage device with ϕ_{sto} of 20 years, but is only checked and repaired every 360 days. The decision of which alternative to choose will depend on the cost of the storage device versus the cost of more frequent detections.

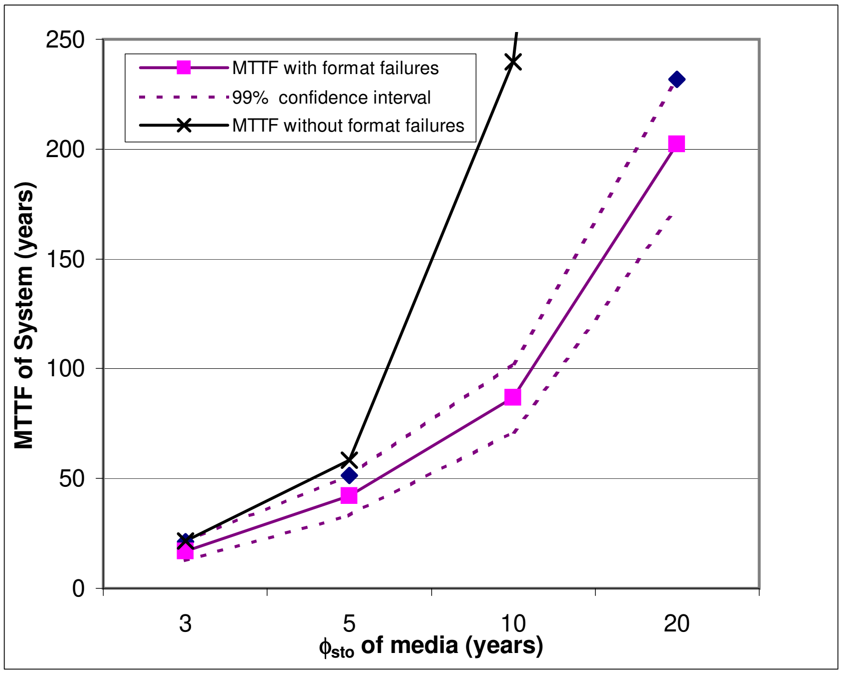

Figure 2.16: MTTF ϕ_{form}=20 years, ρ_{sto}=ρ_{form}=60 days

We now expand our experiments by allowing formats to fail. In Figure 2.16 we show system MTTF as a function of ϕ_{sto} when formats can fail. We fix ρ_{sto} at 60 days, and now formats can fail with $\phi_{form} = 20$ years. To avoid introducing additional factors in the analysis, we assume site failures still cannot happen. Format failures are detected and repaired with $\rho_{form} = 60$ days. For comparison purposes, we have included in Figure 2.16, the results presented in Figure 2.14. The important conclusion that we can derive from the figure is that in an archival repository we cannot focus on single component types. It is surprising that even though the format MTTF is much larger than the disk's MTTF, the failure of formats still has a significant impact. This is because a document is lost if there is a disk failure *or* a format failure. The result is that we are taking the "worst" of those two failures, resulting in a system with a low MTTF. The result of this experiment shows that we need a comprehensive model, like the one proposed in this chapter, to realize the interactions between the components and their effects on system MTTF.

Our model can be used to explore other possibilities that may improve reliability. For example, we now consider what happens if we are able to increase the number of copies maintained in the sites from two to three. In Figure 2.17, we maintain the same parameters at the same base values, but now each site has three copies in three different formats. In the figure, we can see that by increasing the number of copies to three, the MTTF increases from 34 years to 2101 when ϕ_{sto} is equal to 3 years. Although increasing the number of copies to three will undoubtedly increase the cost (as we need an additional 33% disk space and the need to handle an extra format), we have achieved an important improvement in the MTTF of the system.

As we stated earlier, a comprehensive model is important to get an accurate picture of the reliability of the system. In the next experiment, we use our system to explore a different, more complex failure distribution for storage devices. In this new distribution, we want to include the issue of "infant mortality." We will use a failure distribution with a MTTF between 44 and 285 days in the first 30 days, and 5 years afterwards. In this experiment we will assume that formats cannot fail. All other assumptions and repair procedures of the previous experiments are maintained (see Figure 2.13). In Figure 2.19 we show the system MTTF at given percentages

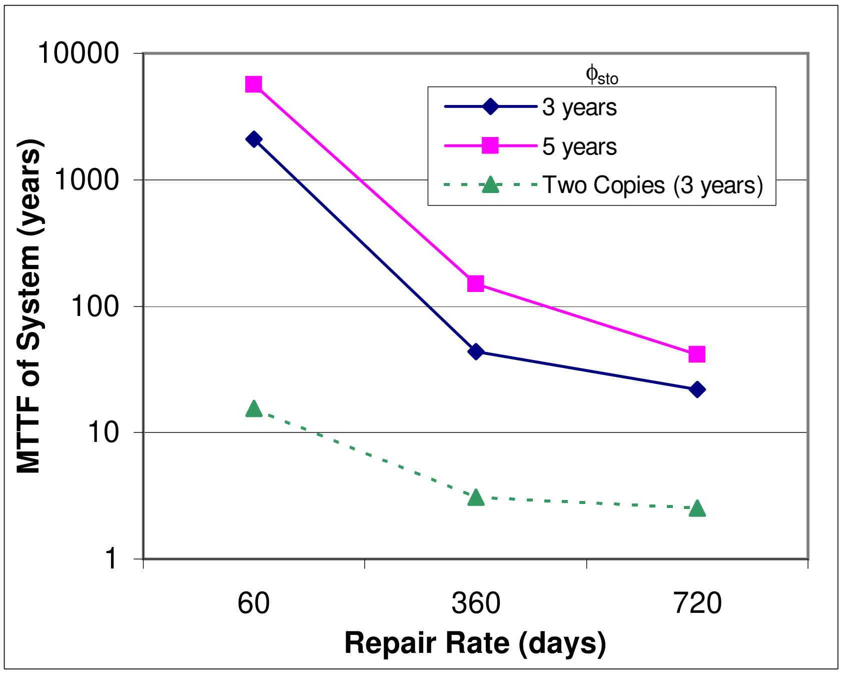

Figure 2.17: MTTF with 3 Copies (logarithmic scale)

of storage devices that fail in the first 30 days. For example, with an early MTTF of 285 days, 10% of the devices will fail within 30 days. With a MTTF of 135 days, 20% will fail. We can convert from one measure to the other by using the formula (conversions shown in Figure 2.18): $\frac{1}{MTTF} = 1 - \sqrt[30]{1 - pct}$

Percentage failed devices	MTTF first 30 days (days)
0%	N/A
10%	285
20%	135
30%	85
40%	59
50%	44

Figure 2.18: Conversion between %failed devices and MTTF

As expected, the higher the infant mortality, the lower the MTTF of the system. At a 0% infant mortality, the system MTTF was 65 years, dropping to 48 years when the infant mortality was 10% and dropping to only 11 years at the 50% infant mortality level. Given this, when using components that suffer from infant mortality, a way to increase the MTTF of the system is for the failure detection module to check new components much more often than older components.

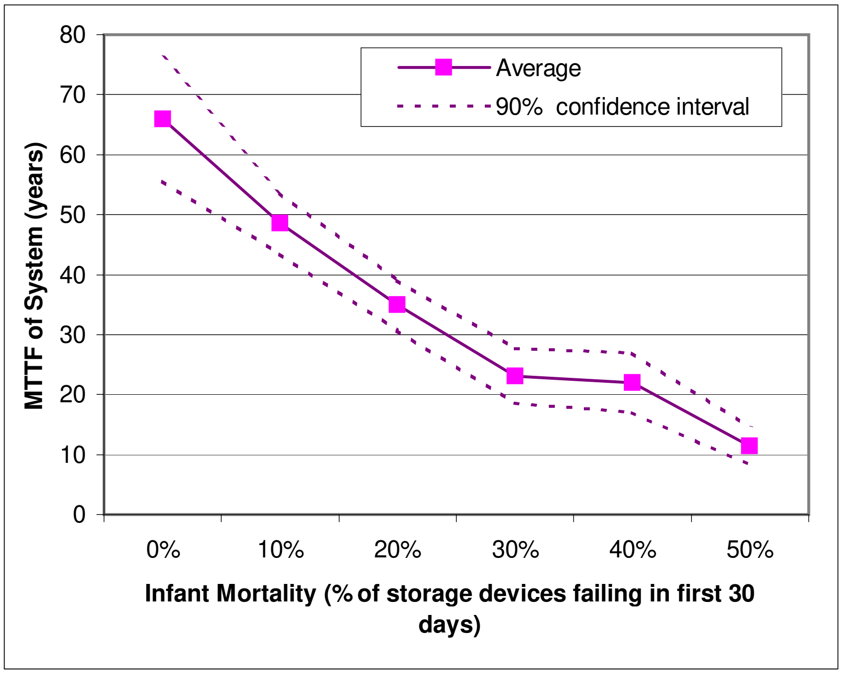

Figure 2.19: MTTF with Infant Mortality

We now explore the issue of the aging of storage devices. With aging, a storage device will have a lower MTTF at the end of its life. For example, a disk may have a 5 year MTTF during its initial life and a MTTF of 2 years when it reaches its "aging" phase. In this experiment we will assume that formats cannot fail. All other assumptions and repair procedures of the previous experiments are maintained (see Figure 2.13). We evaluated the MTTF of the system for different points when aging starts (see Figure 2.20). As expected, the sooner aging starts, the lower the MTTF of the system. If aging never occurs, the MTTF of the system is 65 years. If aging starts after 1 year, the system MTTF is 17 years, increasing to 42 years when aging starts after 5 years. Given this, when using components that suffer from aging, a way to increase the MTTF of the system is for the failure detection module to check old components much more often than newer components. Moreover, we should consider replacing old components with newer ones before the old components fail.

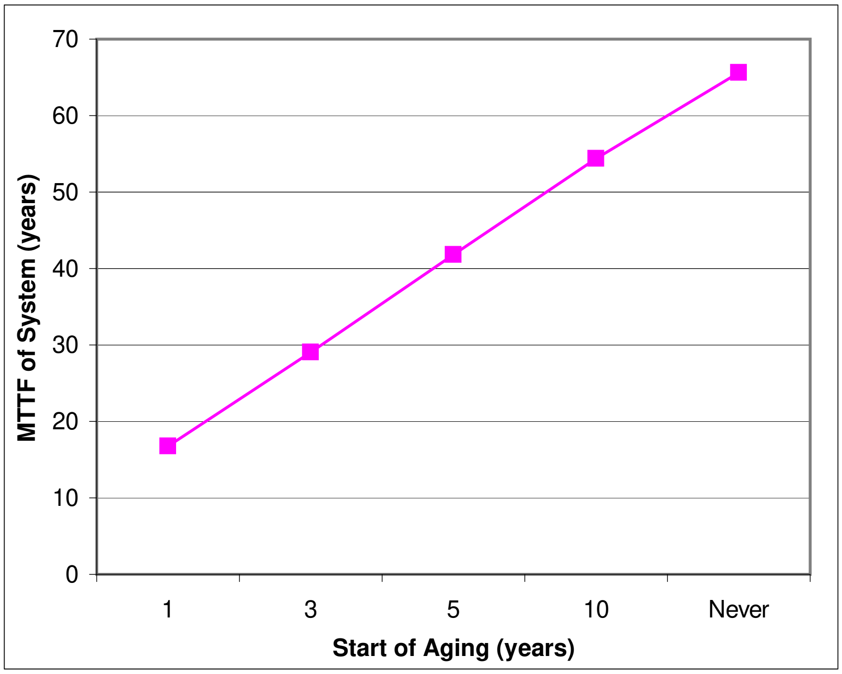

Figure 2.20: System MTTF with Aging and ρ_{sto} of 60 days

As a final experiment, we will evaluate the impact of Preventive Maintenance (PM) on a system with aging disks. Specifically, we will replace old disks with new ones before the old disks are expected to fail. This is done by copying (instantaneously) all documents from the old disk onto a new disk, and then removing the old disk. In this experiment, we are assuming that disks do not have infant mortality and that disks have a 5 year MTTF during their initial life and a MTTF of 2 years when they reach their "aging" phase. Figure 2.21 shows five PM schedules for disks that age at different points. From the figure we can see that the most efficient PM schedule is one that matches the start of the aging period of the disk. For example, when we use a 10-year PM plan a system with disks that age after 5 years, will have a MTTF of 42 years. When we never perform PM, the system MTTF does not increase significantly. However, when we use a 5-year PM plan, the MTTF of the system increases to 63 years. If we keep increasing the frequency of the PM plan, the MTTF does not improve much more.

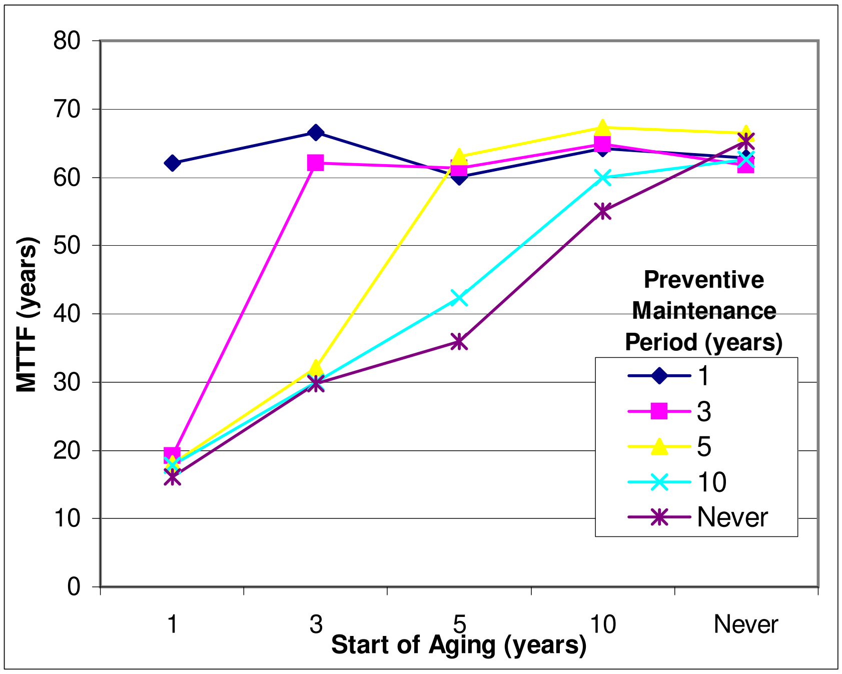

Figure 2.21: System MTTF with PM and Aging

2.7 Discussion

In this chapter, we have studied the *archival problem*. We studied the different options for recovery and preventive maintenance, developing a comprehensive model for an AR. We described a powerful simulation tool, ArchSim, for evaluating ARs and the available archival strategies. We described how ArchSim can efficiently perform large simulations with many components and very long durations. We demonstrated the use of ArchSim in a case study for a hypothetical Technical Report repository operated by Stanford and MIT. We considered options such as disks with different reliability, number of copies, format failure handling, and preventive maintenance. We believe ArchSim can help librarians and computer scientists make rational decisions about preservation, and help achieve better archival repositories.

Chapter 3

A Design Methodology for ARs

3.1 Overview

Two important factors must be considered in AR design: the *level of assurance* (e.g., on average a document will not be lost for 1000 years) and the *cost* (e.g., an initial investment of 1 million dollars and yearly expenses of 100 thousand dollars). In Chapter 2, we studied how to determine the level of assurance of a given system. In this chapter, we expand on that to predict the cost of an AR.

Predicting the cost of an AR is a difficult task. First, we need to estimate the cost of each "event" such as AR creation, the failure and repair of a disk, etc. For many of these events, we may also have to predict when they will happen. For example, since we do not know when a disk will fail, we cannot deterministically predict when and how often we will pay for its repair. Second, we may not know future costs for certain, so we may have to represent them with probability distributions (e.g., the price of a disk may be between \$100 and \$150). As we will see in this chapter, deriving cost estimates and likelihoods for a given AR requires a lot of "guess work." However, the alternative of ignoring costs altogether can easily lead to systems that are overdesigned and overpriced, or that do not meet reliability expectations.

In this chapter we show how AR costs can be modeled, albeit in a rough way, so that rational decisions can be made. In particular, we present a complete design framework for making cost-driven decisions about ARs, and an extension of ArchSim that aids in the process. Our design framework is based on Decision Analysis (DA) theory [35] and we believe that it is a good way of structuring the design of ARs. To illustrate the framework, we use as a running example an extension of the case study presented in Chapter 2.

The contributions of this chapter are:

- An in-depth study of the costs involved in an AR.
- A comprehensive design framework for making AR cost decisions.
- An extended version of our simulation tool that can predict the reliability *and* the cost of an AR.
- A demonstration of the framework in an extended case study.

3.2 AR Costs

We can see the life of an AR as a sequence of events such as the failure of a disk, user access to a document, and making a copy of a document. A *cost event* is an event that has an economic impact. The definition of what has an economic impact will vary from organization to organization. For example, an organization may define economic impact as anything that has an impact on the accounting books (e.g., expenses and depreciation of capital equipment). Another organization may extend the definition to include expenses incurred by the users (e.g., expenses because of unavailability of the system). Cost events may or may not be triggered by a physical event. For example, an organization may buy a maintenance contract for disks under which an annual fee is paid in exchange for free repairs of all disks that may fail during the year. In this case, the failure of a disk (a physical event) will not have any economic impact (and thus it is not a cost event), while the annual payment for the maintenance contract (which is not an AR physical event) will be a cost event.

How can we compare the total cost of two ARs? Ideally, we would like to assign a monetary value (e.g., dollars) to each event in the sequence, and then, aggregate those costs into a single value. Having done that, we can simply choose the system with the lowest cost. If we know the sequence of costs events and each future cost can be deterministically computed (e.g., disk prices will decrease by 5% annually from current prices), this is a feasible task. In this case, we compute the monetary value of each event and we aggregate them by computing the average annual system cost, or ASC (e.g., the AR will cost \$100,000 annually).

However, as we explained in Section 2.1, we may not know the exact sequence of cost events. In addition, we may not know deterministically future costs and we may have to represent costs as probability distributions. In this case, the system is characterized by a probability distribution of ASCs (e.g., with probability 0.3, the annual cost will be \$100,000; with probability 0.7 it will be \$150,000). Although in simple cases the ASC distribution can be found analytically, in general, we have to rely on simulations to obtain an approximation of the distribution.

With probabilistic ASCs, choosing the best AR is not straightforward. The general problem of choosing between two probability distributions of costs has been studied in [50]. In this chapter, we use the simplest way of selecting the best of two cost distributions, namely, we will choose the one with the lowest mean (average). Given this, throughout the chapter, we will frequently talk about the mean annual system cost (MASC) as a representative indicator for the distribution of the average annual system cost.

3.2.1 A Taxonomy of Cost Sources

In this section, we classify the cost sources in an AR. Our goal is to understand those sources, so we can use them as building blocks for cost events. The problem of classifying cost sources for computer systems has been studied in [26], but we are not aware of any studies of the specific case of ARs. The most common cost sources in an AR can be broken down into the following categories:

Hardware and Software: This category includes all the expenses (including lease fees) for servers, clients, disks, software, the network, and peripherals. Although, this is the most obvious source of cost for a computer system, it only represents about 20% of the total cost for the system [26]. Usually, it is easy to estimate the cost of the initial hardware and software, as we can use market prices. However, for replacement hardware and software, it is a more complicated process as we need to predict future prices. Moreover, this prediction is often obtained in the form of a probability distribution, based on current trends, as well as possible future technological developments. For example, when predicting disk costs ten years from now, we may conclude that with 60% probability a terabyte will cost $10 or less, with 80% probability $15 or less, and with 99% probability $50 or less.

Non-labor Operational costs: This category includes all the costs (other than labor) necessary to maintain the AR operational. For example, these costs will include the electricity consumed by the system, air conditioning, and physical space. As with the Hardware-and-Software category, it is easy to estimate the initial cost of non-labor operational costs. For future costs, the major challenge in estimating these costs is trying to predict the future needs of the components of the AR. For example, technological improvements may reduce the need for physical space, but they may increase the need for air conditioning.

Labor costs: This category includes all the human-related costs necessary for the AR. In particular, this will include management (e.g., system administrator), support (e.g., help desk), and development (e.g., application developers).

Information acquisition: Information is sometimes free (e.g., technical reports, thesis), but in general, libraries need to pay for information (e.g., journals). This payment may be a one-time fixed cost, periodic payments (subscriptions), or, more infrequently, pay per use. In some context, we may choose to ignore this cost and considered it "the cost of doing business" (i.e., the library will have to provide access to information even if it does not have an AR). However, we should consider this cost if the creation of the AR will change the way the library pays for information (for example, moving from a paper-based library to a digital library with publishers charging different amounts for paper journals than for digital journals).

Insurance: We define insurance as any agreement where an outside party takes the risk of a specific failure in an AR component in exchange for a fixed payment. An example of insurance is a maintenance contract where the library pays a fixed amount to a company that replaces failed disks. Insurance is important not only because of its direct cost, but also because it can reduce the *variance* of the AR cost. If we are able to "insure" all uncertain events, then we will have a deterministic ASC.

Unavailability: If the system is not available, there may be an economic impact on the organization. Unavailability may be caused by a system failure, but it can also occur when system resources are diverted to maintenance or repair tasks. For example, a user may be blocked because the system is checking for errors in the storage device that holds the requested document. Similarly, the system may only be able to handle a fraction of the normal users when it is migrating documents to a new format.

Measuring the cost of unavailability is a difficult task. If users pay for access, we may be able to assign a direct cost corresponding to lost income. If users do not pay directly, we still may want to penalize the system for unavailability, lest we end up with a design that disregards user needs. One way is to assume that users would access an alternate system (even if the content is not available elsewhere). We could, for instance, assume that the alternate system is equivalent to our AR. Thus, if it costs \$500 per day to operate the AR, the cost of unavailability would be \$500. We could also consider a commercially available alternate system. For instance, an average search on Dialog (SciSearch database) costs \$6, so if we cannot satisfy 1500 requests while doing preventive maintenance, the additional cost would be \$9,000.

Cost of losing a document: Even though our objective is to preserve all documents in the repository, in some circumstances one can put a price on document loss. For example, an organization may choose between archiving certain documents or recreating them. In this case, the cost of losing the document would be the cost of recreating it. Of course, there are cases in which we cannot put a dollar value on losing a document (e.g., the diary of a famous person), so we can use an arbitrarily large cost.

3.2.2 A Taxonomy of Cost Events

In the previous section we studied cost sources. In this section, we use those sources as building blocks for the most common AR cost events. Cost events can be broken down into the following categories.

AR creation: Starting an AR involves a large number of expenses. Hardware and software need to be bought, infrastructure needs to be put in place, new personnel needs to be hired, and so on. For instance, the creation of an AR with 100 disks would involve a server (about \$5,000), the disks (\$500 per disk for a total of \$50,000), installation costs (one consultant at \$1,000), renting and furnishing office space (\$800 for the realtor that finds the place and \$2,000 for furniture and other necessary improvements for the rented space), and loading of the documents (five days of work supervised by a system administrator, about \$1,200) for a total of about \$60,000. The AR creation cost can be amortized over time. Amortization can done by either charging a fraction of the startup cost over fixed periods of time (in which case it would be an operational cost) or over each usage of the system (in which case it would be a document access cost).

Document Access: When accessing a document, the AR may incur acquisition costs or labor costs (e.g., the cost of the operator who retrieves and mounts a tape).

AR operation: The total operational cost of the AR would include the office space taken by the repository, the necessary utilities (electricity, network, etc.), and the cost of the people in charge of keeping the system running. For example, in San Francisco, the average cost of office space is \$380 per square meter per year, so if we assume the repository occupies a small office of $8m^2$, the annual space cost would be \$3,040 per year. Reasonable estimates for utilities are \$4,000 for electricity and \$3,000 for network connectivity. Finally a quarter-time system administrator and a 1/8th librarian would cost about \$20,000 per year. This results in a total operational cost of about \$30,000 per year.

Failure Detection: To enhance reliability, an AR needs to periodically check for failed components (e.g., corrupted tape). When performing failure detection, we should not only take into account the cost of the detection itself (e.g., moving a tape from storage, mounting the tape on the reader, checking the tape, and returning the tape to storage), but also the cost of the unavailability that it may generate. In Section 3.3.4 we will see a specific example of how to compute these costs.

Repairs: When the AR fails and needs to be repaired, cost events may be generated (if we do not have a maintenance contract). For example, when a hard drive fails, we may need to buy a new hard drive (about \$500), remove and install the new one (\$100 for the time of the technician), and restore the contents of the failed drive onto the new disk (\$200 for the network cost and the unavailability caused by the transfer).

Preventive Maintenance: Before a component fails, we may want to transfer the information to a new component. For example, if we know that tapes can survive 20 years, we may decide to copy old tapes onto new ones after 10 or 15 years. The cost associated with a preventive maintenance event includes the cost of the new media, the transfer of the information, and the possible unavailability that this task may create in the AR.

Upgrades: Upgrades are similar to preventive maintenance, i.e., we transfer information from old components to new ones. However, the motivation and the cost implications of an upgrade are different. We perform upgrades to obtain some advantage from modern technology. These advantages may go beyond improved reliability (which is the reason for preventive maintenance) and may include reduced cost. For example, when upgrading to modern hard drives, we may gain reduced operational costs (e.g., if they require less administrator time, less power, or less physical space). Therefore, after an upgrade, we need to reconsider all other costs in the system and change the cost events appropriately.

3.3 Designing an AR

Our goal is not simply to evaluate a given AR, but instead to design an AR that meets our cost and reliability targets. For instance, we need to decide how many document copies to keep, what formats to store them in, how frequently to check for errors, and so on, in order to attain some desired reliability and maximum cost. To aid the design, we use a framework based on Decision Analysis [36]. We show our design framework in Figure 3.1. The framework is a cycle where we first formulate our objectives. Then, we identify the uncertainties (e.g., when a disk will fail). A large number of uncertainties can make the system difficult to analyze, so we next identify and eliminate the uncertainties that do not have a critical influence on the overall performance of the system. Then, we assess the probability distribution of the uncertainties and predict the performance of the AR design. Finally, we perform a sensitivity analysis to appraise our design. If we find a problem with the recommended design (for example, we discover that one of our initial assumptions is incorrect), then we iterate over the cycle.

To illustrate the design process, we will use an extended version of the case study presented in Chapter 2. In this case study, we design an AR of Stanford and MIT technical reports with emphasis on cost implications. At each step, we will discuss the assumptions or decisions made in this sample design. We will also present simulation results for this case study to show the types of conclusions that can be reached.

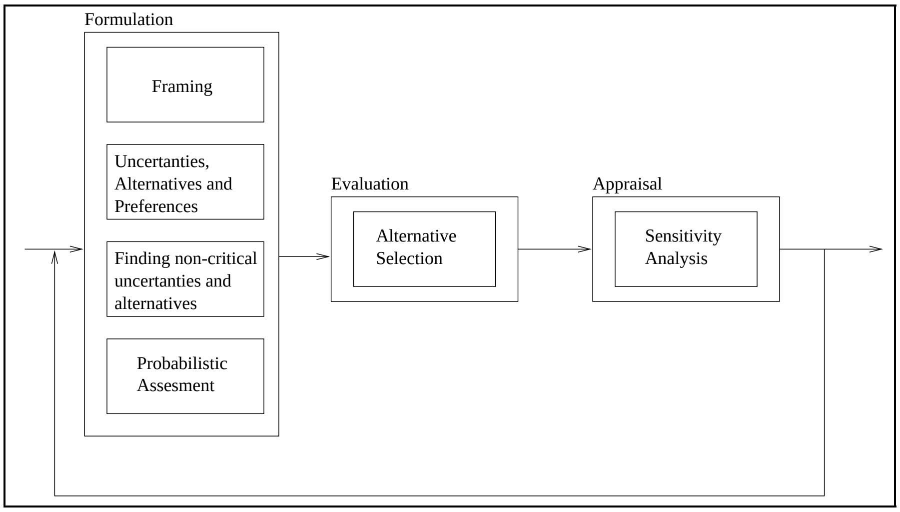

Figure 3.1: AR Design Cycle

3.3.1 Framing the problem

The first step is to clearly define the success criteria for the design. Typically, the criteria will include the archival guarantees (MTTF) and the cost of the AR (MASC). Possible goals can be: (i) Maximize the MTTF of an AR such that the MASC is less than a given amount. (ii) Minimize the MASC of an AR for a minimum MTTF. (iii) Maximize some combination of the MTTF and MASC. In other words, we want to transform the MTTF and MASC to a common metric (let us say dollars) and maximize its combination. For example, if documents can be recreated, the organization may be able to assign a dollar value to losing a document as we discussed in Section 3.2.1.

In our case study, the goal will be to have a repository with a MTTF at least equivalent to that of standard paper (100 years) with the minimum MASC possible. A failure is defined as the loss of one or more documents.

When designing an AR, some decisions are made before starting the design process (policies), others are delayed until the implementation of the system (tactics), while the rest are the focus of the design (strategies). For example, in our case study we assume that the AR will cover Stanford and MIT technical reports (a policy) and that the decision on the specific brand of the hard drives that the AR will use can be deferred until implementation time (a tactic). It is important for the design team to agree on which decisions are policies or tactics, as no time should be spent studying them during the design process.

3.3.2 Identifying Uncertainties, Alternatives, and Preferences

Uncertainties are probabilistic factors that affect the AR (e.g., the time when the disk will fail). Despite their name, we might actually have some control over an uncertainty. For instance, even though we do not know when a disk will fail, we may be able to choose between disks with different MTTFs. When we control the value of an uncertainty completely, we call it a *variable*. For example, if we assume that disk prices will decrease exactly 5% per year (and we know the current price), then the cost of a replacement disk becomes a variable.

Alternatives are the different designs that we have available. For example, in our case study, we may consider:

- ARs with disks with MTTF of either 3, 5, 10, or 20 years.
- ARs with failure detection intervals of 30, 60, 120, or 720 days.

The combination of these different values results in 16 possible configurations that we need to evaluate.

3.3.3 Modeling an AR

An important decision is the level of granularity in the model. If we have too little granularity, then we will have complex uncertainties that are difficult to analyze. If we have too much granularity, the number of variables will be high, making the analysis of the model difficult and even impossible. For example, in our case study we decide to use disks, sites, and formats as the lowest level of detail (in contrast to choosing documents, files, or even bits). Thus, we only need to quantify how much money disks, sites, and formats will cost and how they will affect MTTF. This is much simpler than trying to find the MASC and MTTF of the AR as a whole (not enough granularity), or the MASC and MTTF of every single file (too much granularity).

To model and evaluate a particular AR configuration, we propose an extension of the model presented in the previous chapter. In particular, our extension adds cost events and their associated cost distributions. Recall that our model of an AR has two major elements: a non-fault-tolerance data store and an archival system (AS) that ensures long-term survivability of the information. To model the store, we need to define (see Chapter 2 for a detailed description of these items):

- How many component instances and types are present in the system: that is, how many disks, formats, etc., are available.
- Time distributions for component failures.
- Time for performing a component check.
- Time for repairing a component failure.
- Failure interdependency graph.

To model the AS we need to define for each component not only the distributions that affect reliability (which were described in Chapter 2) but also their cost distributions:

- Document Creation algorithms and their associated cost distributions.
- Document Access algorithms and their associated cost distributions.
- Failure Detection algorithms and their associated cost distributions.
- Damage Repair algorithms and their associated cost distributions.
- Failure Prevention policies and their associated cost distributions.
- Upgrade algorithms and their associated cost distributions.

Figure 3.2 summarizes the AR model for our case study. The failure and cost distributions for the model are described in the next subsections. Note that for simplicity the model assumes no format or site failures. (Our methodology can of course handle a more general model.)

3.3.4 Transforming Non-Critical Uncertainties into Variables

We can simplify the AR analysis by considering as variables the uncertainties that have little impact on MTTF and MASC. For example, if the distribution for disk prices introduces little variation of the total cost, we might as well replace it with its mean. Eliminating uncertainties can save substantial analysis and simulation effort. We call uncertainties that have a large impact on MTTF or MASC the *critical uncertainties*. The remaining ones are called *non-critical uncertainties* or, given that we are fixing them, just *variables*. In this subsection we will see how can we identify critical and non-critical uncertainties.

To determine the impact of an uncertainty, we need to find its distribution. Obtaining an exact probability distribution for each uncertainty may take a significant effort with a limited payoff, so instead we approximate the distributions by using just three values: low, base, and high which correspond to the distribution 10, 50, and 90 percentile. Finding the appropriate low, base, and high values for an uncertainty is more an art than a science. Only experience and a good understanding of the AR components allow one to make these predictions.

After approximating the distributions of the uncertainties, we assess their impact by using a Tornado Diagram. A Tornado Diagram shows the system performance (MTTF or MASC) for the low/base/high value of each uncertainty (while keeping all other uncertainties at their base values). An example of a tornado diagram can be found on Figure 3.3. We will explain this diagram in detail later in this section, but for now, we can see that some uncertainties (such as the Disk Failure) impact MTTF significantly while others (such as Failure Detection Cost) have little or no impact.

Returning to our case study, let us consider the case in which disks have a MTTF

- **AR Description**
  - Initial collection: 200,000 documents. No documents created after startup. Each document, d, will have materializations:
    - $\langle d, MIT, disk_i \rangle$,
    - $\langle d, MIT, disk_k \rangle$,
    - $\langle d, Stanford, disk_x \rangle$,
    - $\langle d, Stanford, disk_y \rangle$.

    Where MIT and $Stanford$ are the two sites; and $disk_i$, $disk_k$, $disk_x$, and $disk_y$ are different storage devices.
  - Number of components and types: 100 storage devices in each site, 2 sites.
  - Failure dependency graph: $site \rightarrow disk$, when the disk is in the given site.
- **Policies**
  - Document Creation policy: for each document, two materializations are created, one in each site.
  - Document to Materialization: read from any materialization.
  - Failure detection algorithm: complete scan of all disk. Site failure detection is instantaneous.
  - Damage Repair algorithm: discard bad component and replace with new component instantaneously.
  - Failure prevention algorithm: none
  - Upgrade policy: none
- **Distributions (unknown for now)**
  - Disk Failure dist. during access (time)
  - Disk Failure dist. during archival (time)
  - Disk Failure Detection success dist. (probability)
  - Disk Repair success dist. (probability)
  - Disk Failure Detection interval dist(time)
  - AR Creation Cost. (dollars)
  - AR Operational cost dist. (dollars)
  - Disk Failure Detection cost dist. (dollars)
  - Disk Repair cost dist. (dollars)

Figure 3.2: Archival Repository Model Parameters

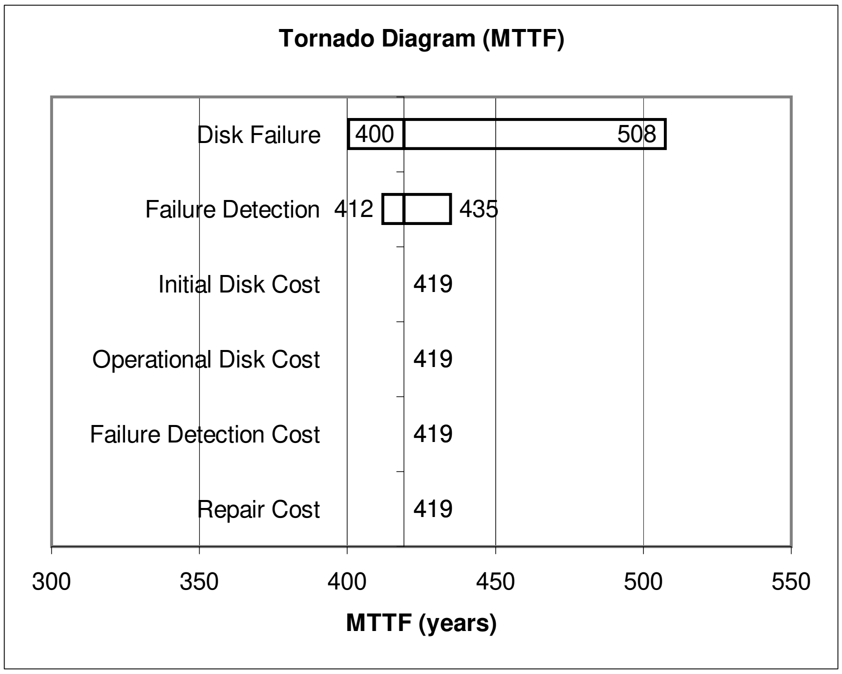

Figure 3.3: Tornado Diagram (MTTF)

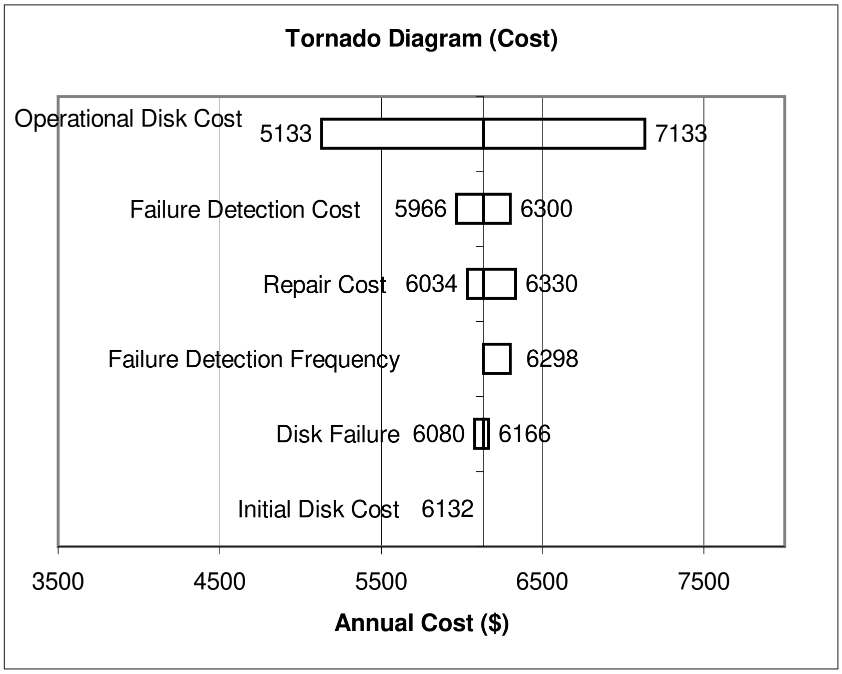

Figure 3.4: Tornado Diagram (Cost)

Variable	low	base	high
Disk MTTF during access (years)	20×0.9	20	20×1.2
Disk MTTF during archival (years)	20×0.9	20	20×1.2
Success of a Failure Detection (prob.)	1	1	1
Success of a Failure Repair (prob.)	1	1	1
Failure Detection Interval (days)	120×1.5	120	120×0.9
AR Creation Cost (dollars)	55000	60000	70000
AR Operational Cost (dollars/year)	200	300	400
Failure Detection Cost (dollars/run)	$1000 + 1200 * 6$	$1200 + 1500 * 6$	$1400 + 1800 * 6$
Repair Cost (per replaced disk)	$450 + 100 + 164$	$500 + 100 + 204$	$600 + 100 + 244$

Figure 3.5: Base values

of 20 years and we scan the repository every 120 days. (In practice we would do a similar evaluation for each of the other 15 alternatives discussed in Section 3.3.2). First, we obtain a *rough* range for the values of the variables. These ranges are shown in Figure 3.5. The choice of these values is highly subjective, but, nevertheless, we will attempt to describe the rationale that an expert may have followed to reach these values.

Disk failure during access and archival: For these two uncertainties, we choose to have the same distributions, since disks do not fail significantly more when accessed. We use as base value the MTTF advertised by the manufacturer (20 years). We assume that there is little variation in MTTF, so we will assign a low value of 90% of the advertised MTTF and a high value of 120% of the advertised MTTF.

Success of failure detection and repair: For these two uncertainties, we assume that the probability of success is 1. In other words, we are assuming that there are no hidden failures (i.e., if a disk is defective, we can always tell) and that all repairs are successful (i.e., we can always replace a defective disk with a working one). Note that the latter does not mean that we can always repair a document. It just means that we are always able to install a new disk and to copy the contents of the failed disk from alternative sources *if it is available.*

Failure Detection Interval: This uncertainty shows that the assignment of low, base, and high values need not be symmetrical. For instance, we assigned a low value of 1.5 times the targeted mean time to detection (120 days), while we assigned a high value of 0.9 times the detection time. In other words, it is more likely that detection will be slower rather than faster.

AR Creation Cost: Using the rationale presented in Section 3.2.2, we estimate the initial AR cost to be \$60,000. To allow for error, we choose a low value of \$55,000 and a high value of \$70,000. We assume that this will be a one-time cost (i.e., no amortization will be done over time).

Operational Cost of a Disk: As illustrated in Section 3.2.2, we use a total operational cost of about \$30,000 per year with a low value of \$20,000 and a high value of \$40,000 per year to allow for errors. This total operational cost divided by the number of disks (100 per site) results in an operational cost per disk of \$200 to \$400.

Cost of the Disk Failure Detection Algorithm: This is probably the hardest uncertainty to estimate. We divide the cost of the detection algorithm into two components. The first component represents the direct cost of running the algorithm, while the second component reflects the cost of service unavailability. The direct cost of the failure detection algorithm includes the time required by the System Administrator to start the scan and correct any problems with the scan (assuming these tasks are not included in the administrator's salary already). If we assume that failure detection involves 5 days of part-time work, the cost will be about \$1,200 per run (see Section 3.2.2).

To estimate the unavailability cost, we assume that users would use an alternate commercial service. Using the costs of Section 3.2.2 for 1500 missed user requests, we price unavailability at \$9,000. Therefore, the total cost of running the failure detection algorithm is about \$10,200. To allow for error, we choose a low value of \$8,200 and a high value of \$12,200 per run.

Cost of the Disk Repair: The repair cost is equal to the cost of adding a new disk (\$450 to \$600) plus the cost of removing the disk (\$100) and a fixed amount for the resources involved in copying the data from the alternate sources onto the new disk (equal to twice the cost of running the detection algorithm on a single disk, that is, for the base cost, $\$10,200/100 * 2$ or \$204).

We are now ready to generate the Tornado Diagrams. We use ArchSim to simulate the performance of the system. At this stage, we do not want to run the full fledged simulations (which may take several hours). Instead, we run *fast* simulations with broad confidence intervals (e.g., 90%) that require fewer repetitions and consider all uncertainties, except disk failures, to be deterministically fixed at their base values. Fixing the value of the variables speeds up the simulation as we do not need to compute a random value for each event and allows us to group events. For instance, instead of generating a random value for each repair cost, we simply count the number of repairs that were performed during the simulation and multiply by the fixed cost of making a repair. We treat disk failures differently because deterministic failure times would cause all disks to fail at the same time (and all data would be lost).

To generate the Tornado Diagrams, we evaluate the AR reliability and cost for the proposed design with all the variables at their base values. Then, we modify each variable independently (while keeping all others at their base value) to its high and low value and evaluate performance again. Each tornado diagram summarizes $1 + 2 \times$ variables simulations (one simulation for the base case and two for each variable). In our case, this results in a total of 13 simulations per tornado diagram. We show the result of our simulations in Figure 3.3. We can see that most of the MTTF variation (95.7%) comes from the disk MTTF variation. Therefore, with respect to this metric, we can safely assume that the other uncertainties are noncritical and can be fixed at their base values.

Figure 3.4 shows the equivalent diagram for costs. In this case, 94% of the cost variation is produced by the operational cost of the disks. Therefore, with respect to this metric, we can safely assume that the other uncertainties are non critical and can be fixed at their base values.

In conclusion, we only need to consider disk failures and disk operational costs as critical uncertainties for the case of a design with disks having a MTTF of 20 years and failure detection interval of 120 days. To complete the analysis, we need to repeat the process with the other 15 configurations. Although we do not show the results for the other cases, the conclusion is the same: only disk MTTF and cost are critical. (In general, the conclusions could vary from scenario to scenario, but this does not occur in our case study.)

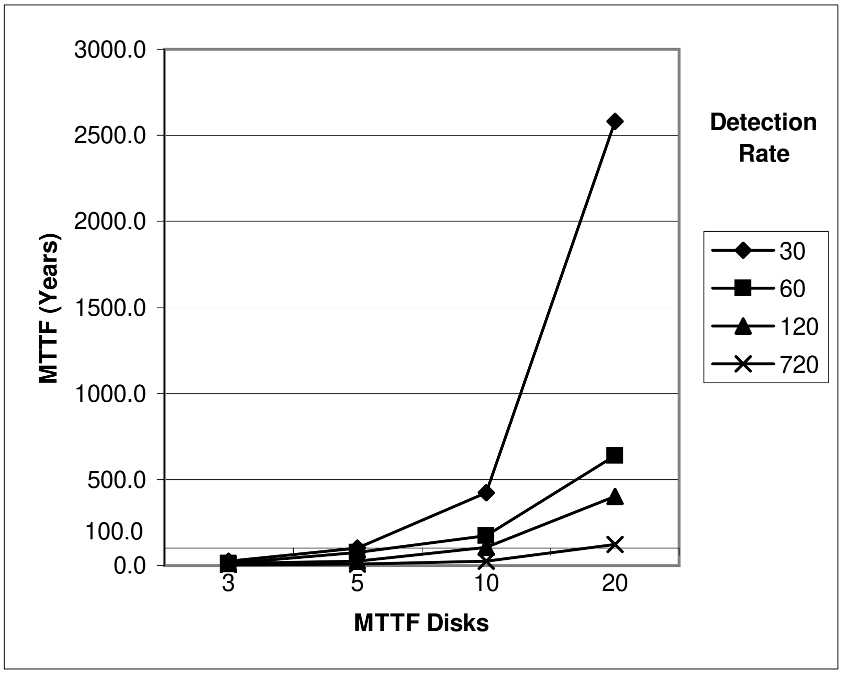

Figure 3.6: MTTF for base values

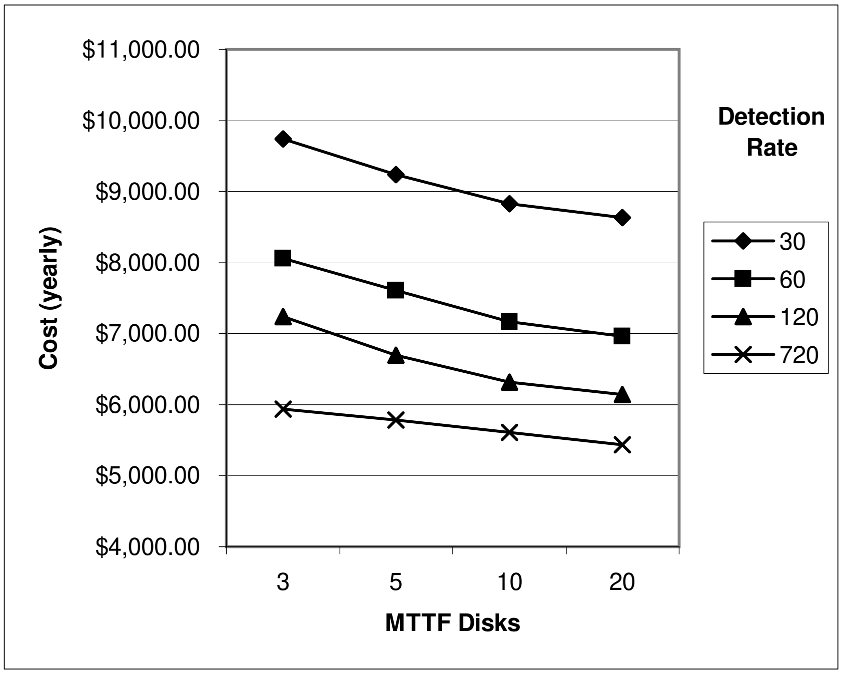

Figure 3.7: MASC for Base Values

3.3.5 Eliminating Futile Alternatives

Let us turn our attention to the available alternatives. For that purpose, we used ArchSim again to run fast simulations. The results are shown in the graphs of Figure 3.6 and 3.7. From the graphs, we can see that a detection interval of 720 days never achieves our required minimum of a MTTF of 100 years (it barely achieves it for disks with a MTTF of 20 years, but the 90% confidence interval includes values below 100 years). Similarly, disks with a MTTF of 3 years also never achieve our required minimum MTTF. Note that these results are based on fast simulation where all uncertainties, except the MTTF of disks, are fixed. If we were very aggressive and eliminated too many alternatives, we might eliminate the alternative that might be the best when running the full simulation. On the other hand, by eliminating some alternatives, the time to run the full-fledged simulations later is reduced. For this case study, we not consider further disks with a MTTF of 3 years or detection time of 720 days. If we were more aggressive, we could have also eliminated disks with a MTTF of 5 years (except when the detection interval is 30 days).

Regarding MASCs, the preliminary analysis shows a surprising result. The MASC of an AR with costly, but more reliable, disks ends up lower than that of an AR with the cheap, less reliable, disks. This is because of the cost of buying a new disk (when the cheap disk fails) and transferring the information to it. Therefore, we drop disks with a MTTF of 3 years, and detection intervals of 720 days, and reduce our alternatives from the original 16 to just 9.

3.3.6 Probabilistic Assessment of Uncertainties

For our final analysis we may need to assess the probability distributions of uncertainties more precisely. In practice, we will rely on experts to produce these distributions. Techniques for probability distribution elicitation are described in [83].

To illustrate, in our case study, we model disk failures with an "infant mortality" distribution. Recall that this kind of distribution has two phases: First, when most manufacturing defects would cause a failure, the probability of failure is high, but

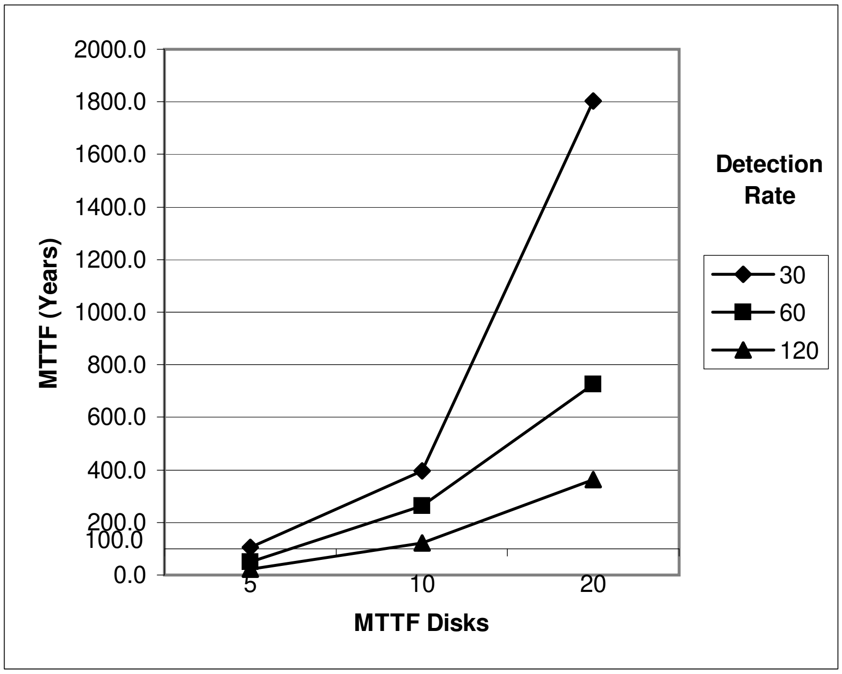

Figure 3.8: MTTF Evaluation

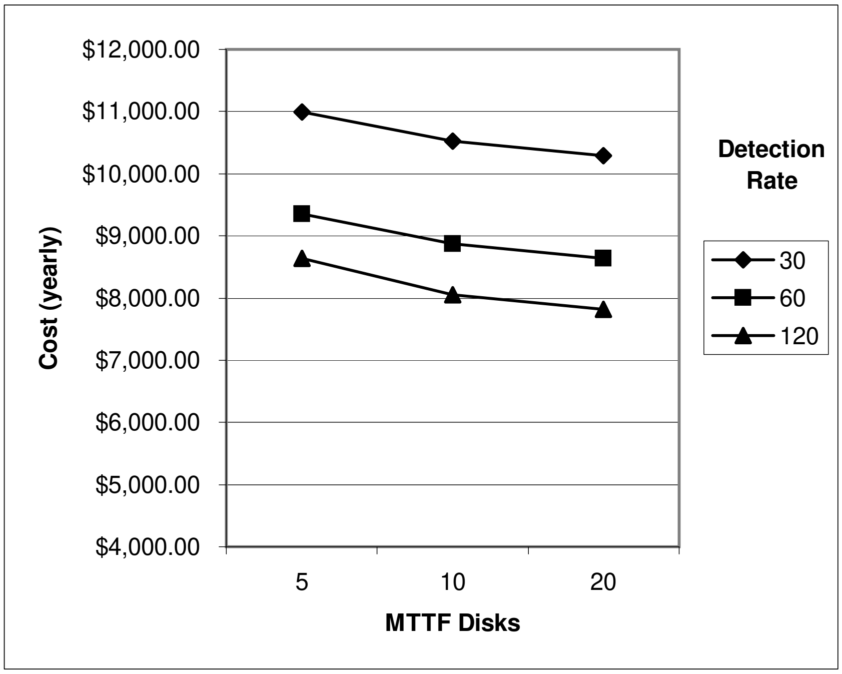

Figure 3.9: Cost Evaluation

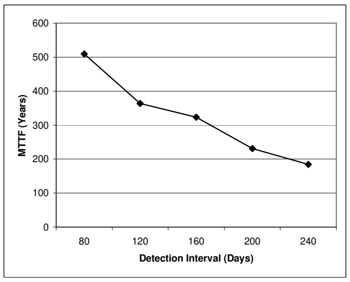

Figure 3.10: MTTF Sensitivity Analysis (Detection Interval)

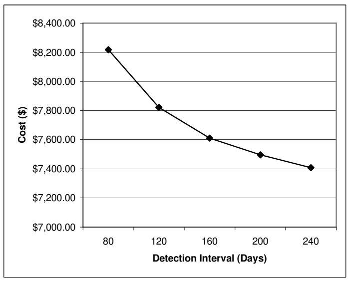

Figure 3.11: MASC Sensitivity Analysis (Detection Interval)

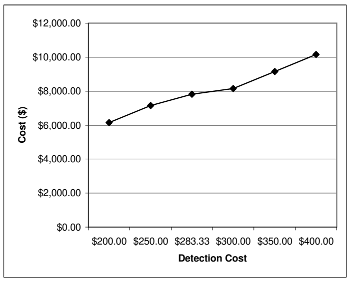

Figure 3.12: MASC Sensitivity Analysis (Detection Cost)

drops sharply over time; then, in the second phase, the probability of failure is constant. For our disks, we use a distribution where 10% of the disks fail within the first 30 days (i.e., an exponential distribution with a mean of 285 days) and, after that, disks fail following an exponential distribution with a mean of 20 years.

To model the operational cost of disks, we assume that the library would sign one-year maintenance contracts. Although the price is fixed for one year, from year to year, the price specified in the contract may change due to market conditions. We use a uniform distribution between \$200 and \$400 per disk to capture these market fluctuations. Using these more complex distributions makes our predictions more accurate, but also makes evaluation much harder. Fortunately, ArchSim can handle such general distributions.

3.3.7 Evaluating an AR Design

In the previous sections, we concluded that the most promising alternatives were the ones with disks with a MTTF of 5 to 20 years and a detection/repair interval of 30 to 120 days. We also concluded that we would consider the MTTF of the disks and the yearly operational cost of the disks as critical uncertainties, and the rest as variables. Using this setup, we use ArchSim to fully simulate the AR and obtain its reliability and cost.

In Figure 3.8 we see the MTTF of the AR for different configurations. From the figure, we conclude that we need disks with a MTTF of either 10 or 20 years and detection intervals of 30 to 120 days to achieve our target MTTF of 100 years.

In Figure 3.9 we see that the least expensive alternative is the one with a detection interval of 120 days. Consistent with the preliminary simulation, here again the cost decreases when using more reliable disks. Therefore, the best alternative is one that uses disks with a MTTF of 20 years and has a detection interval of 120 days. Such an AR will have a MASC of $7,822 and a MTTF of 364 years. Note the critical role that costs played in reaching this decision: if we had ignored costs we could have easily selected a design that would achieve the desired MTTF but in a much more expensive way!

3.3.8 Appraising Cost Decisions

In this final phase, we revisit our assumptions by running a sensitivity analysis of the critical and non-critical uncertainties. We again illustrate the process via our case study. Our proposed design was an AR with a disk with a MTTF of 20 years and a detection interval of 120 days. Using ArchSim we ran sensitivity analyses for all uncertainties, but we will illustrate the results for just two of them: the Detection Interval (a critical uncertainty) and the cost of detecting failures (a non critical uncertainty).

In Figure 3.10 we perform a sensitivity analysis for the detection interval. We want to find out the impact of a small change in the suggested 120-day interval. In the figure, we can see that the smaller the detection interval, the higher the MTTF of the AR. In particular, an AR with a detection interval of 240 days will have a MTTF of 184 years. Thus, we can double the value of the detection interval, and we still achieve the target archival guarantees. If doubling the value of the detection interval had caused an important decrease in cost, then we would have needed to reassess our recommendation.

In Figure 3.11 we see the AR cost for different detection intervals. As expected, larger detection intervals decrease costs. For instance, increasing the interval to 240 days causes a reduction of cost of $400 or about 6%. We now have to decide if it is worth considering new alternatives given a potential saving of $400. If this is the case, we should return to the formulation phase and add alternatives with detection intervals in the 120 to 720 day range. Notice that we cannot make a new recommendation based only on the sensitivity analysis, because the variable we want to change may interact in unexpected ways with the reliability and cost metrics. Concretely, in this case, the cost associated with the detection interval might not be a continuous function, so we may need to revisit our cost estimates and re-run the simulations.

Let us now turn our attention to the sensitivity analysis of the cost of detecting failures. This variable has a different nature than the detection interval, as it does not affect the MTTF of the AR. Additionally, we may not be able to change the value of this variable (e.g., the cost of detecting failures may be determined by the market). Therefore, a sensitivity analysis here rather than validating or invalidating our proposal, gives us an idea of how much the cost of the AR may increase (or decrease) if our estimate of the value of this variable was erroneous.

Figure 3.12 shows the AR cost for different detection costs. As expected, the figure shows higher costs when the detection cost increases. The important observation here is that costs increase almost linearly with a very small slope. An increase of 100% in the detection cost (from 200 to 400), only results in an increase of 33% in the AR cost. This means that a small error in the estimate of the detection cost will not affect the AR much. Unfortunately, it also means that efforts to reduce the detection cost will have small payoffs.

3.4 Discussion

In this chapter we have studied how to make cost-driven decisions about ARs. We presented a framework that improves the efficiency and effectiveness of the AR design process. We believe our design framework and ArchSim can help librarians and computer scientists make rational and economical decisions about preservation, and help achieve better ARs.

Part II

An Architecture and Algorithms for Archival Repositories

Chapter 4

An Architecture for Archival Repositories

4.1 Overview

A digital library *repository* stores the digital objects that constitute the library. The key requirement that distinguishes DLRs from other information stores is archival storage. This archival nature means that the digital objects (e.g., documents, technical reports, movies) must be preserved indefinitely, as technologies and organizations evolve[52, 69].

The goal of this chapter is to present an architecture for the archiving of digital objects. The objective is *not* to replace database systems, but rather to allow existing and future systems to work together in preserving an interrelated collection of digital objects (and their versions) in the simplest and the most reliable way possible. Also, it is necessary to keep in mind that what we are describing is the lowest layer(s) of a DLR; higher layers (not discussed in this dissertation) would deal with intellectual property, metadata, security, and so on.

In Section 4.2 we present the key components of our architecture that make long term archiving feasible. Then, in Sections 4.3 through 4.6 we describe the functional components of the architecture. In Section 4.7, we present a complete example that shows how those components work together. Finally, in Section 4.9 we discuss related work.

4.2 Key Components

Under our architecture, an Archival Repository (AR) is formed by a collection of independent but collaborating *sites*. Each site manages a collection of digital objects and provides services (to be defined) to other sites. Each site uses one or more computers, and can run different software, as long as it follows certain simple conventions that we describe in this chapter. Our architecture is based on the following key components.

4.2.1 Signatures as Object Handles

Each object in an AR has a *handle* used to identify and retrieve it. Handles are *internal* to the AR and are not used by end users to identify documents. (Example:

If a user is searching for report STAN-1998-347-B, a naming facility not discussed here will translate into the appropriate handle, or handles if the report has multiple components.)

Given an object, we define its handle to be a (large) signature computed exclusively from its contents, using a checksum or a Cyclic Redundancy Check (CRC). If the contents are smaller than the size of the signature, the object (at creation time) is "padded" with a random string to make its size larger than the size of a signature. This scheme has the following properties, which are important in an archival environment:

- Each site can generate objects and their handles without consulting other sites. This makes it possible for sites to operate independently. Furthermore, sites only need to agree on the signature function, not on software versions, character sets, timestamp services, and so on.

- The handle for an object can be reconstructed from the object itself. As we will see, this is an extremely useful property, since we do not need to reliably save any handle-to-object mappings.

- If copies of an object are made at different sites, all copies will have identical handles. This may seem disconcerting at first, but if the contents are identical, it makes management simpler to call "a spade a spade."

- Objects with different contents will, with *extremely* high probability, have different handles.

The last item requires some discussion, since it may be possible for two different objects to share a handle, which would be disastrous. However, by making the signature large (e.g., 128 bits or more), the likelihood of this disaster happening is so extremely low that it is not rational to worry about it.

The probability of not having a *signature collision*, p, depends on the size of the collection, n, and the number of bits, b, in the signature. When we insert the first object, the probability of not having a collision is 1 (as there are no documents to collide with); for the second document the probability of not having a collision is

$(2^b - 1)/2^b$ as there are 2^b possible signatures that can be generated and all but one of them will not create a collision. In general, when we have inserted k documents, the probability that the next document will not create a collision is $(2^b - k)/2^b$ if $k <= 2^b$, or 0 otherwise. In conclusion, if we assume that the signature function uniformly distributes documents in the signature space, and that the computation of each document signature is independent, then the probability that we will not have a collision in a collection of n documents is:

$$p = \prod_{k=0}^{n-1} \frac{2^b - k}{2^b} = \frac{2^b!}{(2^b - n)!2^{bn}} \tag{4.1}$$

Equation 4.1 is impractical to use when b and n are large numbers as the factorials will produce an overflow. We can derive an approximation by making $p = \prod_{k=0}^{n-1} 1 - \frac{k}{2^b}$, and using the Taylor expansion for the exponential function to obtain $p \approx \prod_{k=0}^{n-1} e^{-\frac{k}{2^b}}$. Solving the product, we obtain:

$$p \approx e^{-\frac{n(n-1)}{2^{b+1}}} \tag{4.2}$$

Using these equations, we derive a bound for the probability p that there is no disaster in an AR with n objects and signatures of size b bits. The bound is extremely conservative, but yet we see that a 256 bit signature can make even an AR with 10 billion objects incredibly safe.

Collection Size (n)	Probability of no collisions (p)	Signature Size (b)
10^7	$1 - 10^{-9}$	76 bits (10 bytes)
10^8	$1 - 10^{-9}$	83 bits (11 bytes)
10^9	$1 - 10^{-9}$	89 bits (12 bytes)
10^7	$1 - 10^{-24}$	128 bits (16 bytes)
10^7	$1 - 10^{-63}$	256 bits (32 bytes)
10^{10}	$1 - 10^{-18}$	128 bits (16 bytes)
10^{10}	$1 - 10^{-57}$	256 bits (32 bytes)

Figure 4.1: Number of bits required for typical n, p

If some applications (or paranoid users) need an absolute certainty that each signature is unique, then we offer the following enhanced identification scheme. Handles are extended to have two fields: a unique publisher field and the signature of the object. The publisher field is the unique code of the site that first publishes the object; this publisher code is assigned to the site by some authority. The publisher field of an object does not change when the object migrates to other repositories. The second field is the same as the signature described earlier. When a site creates a new object, it first stores its publisher field in the object header. Then it computes the signature of this extended object and checks if any other *local* object has the same signature. In the extremely rare case that there is a conflict, we add a *discriminator*, a random string of bytes, at the end of the new object. The discriminator is included in the computation of the signature (and therefore will make the object map to a different signature), but it is filtered out when the object is returned to a user. From then on, the handle of an object is computed (at any site) by reading its publisher value and adding to it the object signature.

4.2.2 No Deletions

Because of our handle scheme, objects cannot be updated in place. That is, if the content of an object is modified, it automatically becomes a new object, with a different handle. This is actually an important advantage, since it eliminates many sources of confusion. For instance, one cannot correct a typo in a report and pass it off as the same object. (We do provide a higher level mechanism for tracking versions of an object; see Section 4.5.) Similarly, if a stored object is corrupted due to a disk error, the corrupted object will not be confused with the original.

Another fundamental rule in our architecture is that objects are never (voluntarily) deleted. Allowing deletions is dangerous when sites are managed independently; in particular, it makes it hard to distinguish between a deleted object and one that was corrupted ("morphed" into another) and needs to be restored. Ruling out deletions is natural in a digital library, where it is important to keep a historical record. Thus, books are not "burned" but "removed from circulation." We can provide an analogous high level mechanism for indicating that certain objects should not be provided to the public.

Having immutable objects presents some management challenges. For example, say we create a new version Y of some object (a video clip) X. We cannot mark X directly to indicate there is a new version Y that should be accessed, because this would be an in-place update to X. In Section 5 we show how we can "indirectly" record such changes. Of course, having no deletions increases storage requirements. We do not believe this is an important issue because (1) storage costs are so low, and (2) we are only archiving in this fashion library objects, not all possible data.

4.2.3 Layered Architecture

Since each AR site may be implemented differently, it is important to have well defined and as simple as possible site interfaces. Furthermore, it is also important to have clean interfaces for services *within* a site, so that different software systems could be used to implement individual components. We achieve this in our architecture by defining *service layers* at each site. The layers include:

1. *Object Store Layer:* The Object Store layer uses a *Data Store* (e.g., file system, database management system) to persistently save objects. This layer may use its own scheme to identify objects (e.g., file names, tuple-ids). We refer to these local identifiers as *disk-ids*.

2. *Identity Layer:* This layer has two main functions: (i) it provides access to objects via their handles (signatures); and (ii) it provides basic facilities for reporting changes to its objects to other interested parties.

3. *Complex objects layer:* Manages collections of related objects. Its services could be used to maintain the different versions (or representations) of a document.

4. *Reliability layer:* Coordinates replication of objects to multiple stores, for *long term* archiving. The assumption is that the Object Store layer makes a reasonable effort at reliable storage, but it cannot be counted on to keep objects forever.

5. *Upper layers:* Provide mechanisms for protecting intellectual property, enforcing security, and charging customers under various revenue models. It can also provide

associative search for objects, based on metadata or contents of objects, as well as user access.

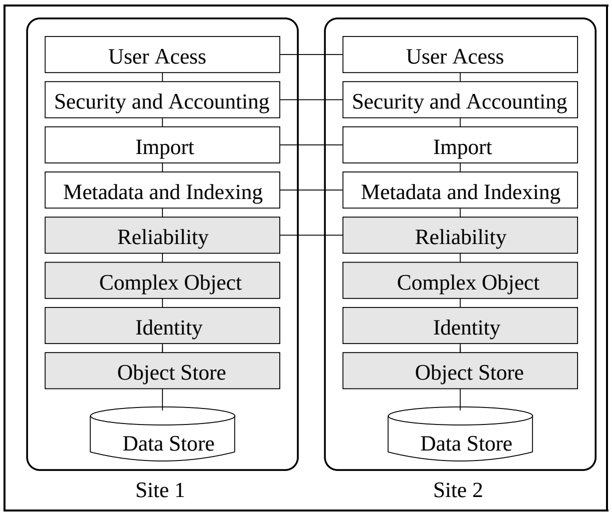

Figure 4.2: Layers of a Cellular Repository.

In Figure 4.2 we illustrate the layers of an AR. Each "column" in the figure represents a site, and each "row" a software layer. We call the implementation of a layer at a site a *cell*, and the complete repository a *cellular* AR. Cells can collaborate with others to achieve their goals. For example, the reliability cell at Site 1 communicates with the reliability cell at Site 2. Cells below the reliability layer only deal with their local site. In this dissertation we only study the grayed-out cells.

4.2.4 Awareness Everywhere

Awareness services (standing orders, subscriptions, alerts) are important in digital libraries. They are also important for our reliability and indexing layers: if one site backs up another, it must be aware of new objects or corrupted objects in order to take appropriate action. Similarly, to maintain an index up-to-date, changes need to be propagated. In many systems, awareness services are added as an afterthought, once the base storage system is developed, and this makes it hard to detect *all* changes. In our architecture, awareness services are an integral part of every layer. This makes it possible to build very reliable awareness services that can be used for replication and indexing.

4.2.5 Disposable Auxiliary Structures

Layers typically maintain auxiliary structures for improving performance. In our architecture these structures are designed to be *disposable*, so they can be reconstructed from the underlying digital objects. To illustrate, let us consider the Identity layer. For efficient lookup, it needs an index structure that maps a handle (signature) into the local disk-id (e.g., file name). One option would be to store this index as a digital object, making it part of the AR. However, this opens the door for inconsistencies. For instance, the index may say that the object with handle H can be found at disk-id D, but the signature of the object at disk-id D is not H. Instead, we maintain that no auxiliary structures are part of the AR. (The structures may be on secondary storage that are not part of the AR.) If the structures become corrupted or inconsistent with the AR, they should be deleted and reconstructed from scratch.

In addition to avoiding potential inconsistencies, this approach also makes it easy to migrate objects to a new store, when the old one becomes obsolete. Auxiliary structures, which are typically intricate, do not have to be migrated to the new system. The new system can simply obtain the digital objects and build its own structures, using whatever implementation it desires.

4.3 Object Store Layer

The Object Storage Layer is the lowest AR layer. This layer treats objects as a sequence of bytes and uses a local disk-ids to identify objects. The disk-ids are meaningful only for a specific Data Store, and their format varies from data store to data store. For example, if the Data Store is a standard file system and each object is saved in a different file, the disk-id could be the file name. On the other hand, if *all* objects are saved in a single sequential file, then the disk-id could be the name of that file, the offset into that file, and the length of the object.

4.3.1 Object Store Interface

The interface of the Object Storage Layer has the following functions:

- `OS_Get(disk_id)`:

  Read an object given its disk-id.

- `OS_Put(bag_of_bits):disk_id`:

  Insert a new object in the repository and return the disk-id associated with it.

- `OS_Awareness():list_of_disk_ids`:

  List all disk-ids.

The last function, `OS_Awareness()`, lets a client perform a "scan" of the entire collection. This is the most primitive type of awareness service one can envision. Its simplicity makes it easier to implement an Object Store that is very robust. This awareness service is used by higher layers when they have lost their state, or when they wish to verify their state.

For building a reliable system, one must not only define the *desired events* (what we have done so far in this section), but also the *undesired expected events* [39]. The latter are those events that may occur because of failures, but which recovery mechanisms (at higher layers) will handle. For this layer, the undesired expected events include: (i) `OS_Get()` returning a corrupted object; (ii) `OS_Put()` failing to insert an object (and returning an error); (iii) `OS_Awareness()` not returning the disk-ids of all objects ever inserted with `OS_Put()`).

4.3.2 Object Store Implementation

Having an extremely simple interface (e.g., no deletes, primitive awareness) reduces the number of undesired events that one needs to consider, and makes it possible to build a rock-solid store, with few "moving parts" and few things that can break. In addition, this simple interface allows us to use almost any secondary storage system as a Data Store, including legacy systems.

To illustrate a possible way to build a solid store that supports this interface, consider the following design. Objects can be placed sequentially on a disk (or tape), with a unique pattern separating them. The disk-id would be the disk address of the first byte. To list all handles, we simply scan the disk sequentially looking for the special start-of-object pattern. Since there are no deletes or updates, any object found during the scan is an object to report. Since there are no auxiliary structures (e.g., no i-node tables, no free space tables), there are no structures that can be corrupted. To migrate this collection of objects to a different site, we simply must move this single stream of objects, and nothing else. We stress that this is not the only way to build a cell for this layer, but it is the way we expect it to be built in a good, reliable repository.

4.4 Identity Layer

The Object Identity Layer provides access to objects through their globally unique handles, provides an awareness service based on handles, and attempts to correct some of the failures of its underlying Object Store cell.

In our architecture, digital objects have two components: a header and a body. For example, from the point of view of the identity layer, the body of a digital object contains the bits given to an Identity cell for storage. In the header, the cell can store system data (e.g., size of object). The resulting object (header+body) can then be sent to the object store. Unknown to the identity layer, the body may contain headers added by higher layers (e.g., the `type` field discussed in Section 5). This is analogous to how packets move between network layers, with lower layers adding their own headers. However, unlike network layers, our lower layers do *not* remove headers when returning an object to upper layers. The complete headers, as recorded in the Data Store, must be preserved, so that any layer can compute the signature and verify that it is the correct object. Of course, each layer only interprets its own header, not those of lower or higher layers.

4.4.1 Identity Interface

The Object Identity Layer implements the following functions, analogous to the Object Store functions:

- `IL_Put(bag_of_bits):handle`:

  Creates an object and returns the global handle associated with it.

- `IL_Get(handle):bag_of_bits`:

  Gets an object given its handle.

- `IL_Awareness():list_of_handles`:

  Returns all handles of objects in the repository.

- `IL_Latest(client):list_of_handles`:

  Lists all handles created in the repository since the last time the `client` invoked this function.

The `IL_Put` function is used to create an object. The function receives the data and calls the Object Store layer `OS_Put()` function to save the data on secondary storage. The handle for the new object is computed and returned to the client. The `IL_Get()` function returns the object given its handle. We discuss below how this function can be implemented.

The `IL_Awareness()` function lists the handles of all objects in the local store. The `IL_Latest(client)` function is a specialized awareness service. We do not explain here in detail how it operates, but intuitively, it reports objects created since the last time the `client` invoked this function. It is provided to improve efficiency, since with it clients do not have to be informed of objects they have seen before. Since `IL_Latest()` must rely on auxiliary structures (somehow recording what new objects have not yet been seen by clients) it is not as reliable as the `IL_Awareness()` function that simply scans the Object Store for all objects. In Chapter 5, we discuss options for implementing such an awareness service.

Undesired expected behavior of this layer includes (i) losing some object; (ii) `IL_Put()` returning an error; (iii) the awareness functions not returning all of the handles. The Identity layer should attempt to make the probability of these and other undesired events as low as possible. One way to do this is to check for undesired events of the Object Store layer. Again, note that our architecture significantly reduces the number of undesired events. In particular, the "wrong" object can never be returned by a `IL_Get` call because the fact that it can be trivially checked that the object matches the requested handle. Similarly, we never return a "deleted" object since there are no deleted objects!

4.4.2 Identity Implementation

There are two ways to implement the `IL_Get(handle)` function. The first is to obtain all disk-ids from the Object Layer, and then retrieve each object in turn and compute its signature, until an object whose signature matches the requested handle is found. The second way is to have the Identity layer keep an index mapping handles to disk-ids. The index is initialized with a complete scan of the Object Store, and then it is incrementally maintained as new objects are created. In this case, the `IL_Get(handle)` function simply looks up the disk-id for the given `handle`, and fetch the object from the store. Note that indeed this index is disposable, as discussed in Section 4.2.5. As a matter of fact, in a good implementation, the index will be periodically discarded and rebuilt from scratch, to ensure that its structures have not been corrupted, i.e., to reduce the likelihood of undesired events at this layer.

Similarly, the `IL_Latest()` function uses auxiliary structures to track the objects not yet seen by a client. These structures must also be disposable. It should periodically be deleted, in order to force clients to use the more general `IL_Awareness`. This causes the client to check if it indeed has all the objects known to the Identity layer, and to re-initialize the auxiliary structure used for future `IL_Latest` calls.

As discussed earlier, the Identity layer must handle many of the undesired events of the lower cells. Specifically, suppose that the Identity layer is servicing a `IL_Get(handle)` call, and that through its structures has determined that the object is at `disk-id`. Since the call `OS_Get(disk_id)` may return a corrupted object, the Identity cell must check that the fetched object indeed has the handle `handle`. If there is a discrepancy, the Identity Layer reports that the object has not been found (and perhaps attempt to reconstruct the mapping between handles and disk-ids). However, it cannot restore the object; this service will be provided by the Reliability Layer, discussed later in this chapter. (Actually, we cannot be sure the problem was caused by the Object Store; it could be the case that the auxiliary structure which told us that `disk-id` was the place to look for the object was incorrect.)

4.5 Complex Object Layer

In an AR, multiple digital objects may be interrelated. For example, a technical report may have several renditions (e.g., plain ASCII, postscript, Word97), where each of these is a simple object. Similarly, a report may consist of a sequence of versions, representing the state of the report over time. The Complex Object layer implements three useful constructs, tuples, versions, and sets (among others), that can be used for implementing higher level notions such as "technical report," and "access rights for a movie." In this chapter we do not address the details of the high level concepts, which would be implemented by higher layers. References [21] and [68], among others, propose specific organizations for "documents" and other high level constructs.

Traditional methods for building complex structures do not work in our AR environment because objects cannot be deleted or modified. For instance, we cannot implement a set as an object containing pointers to other member objects, since the membership could never be modified. (If the set represents the renditions of a report, it would mean that a new rendition could never be added, for example.) The schemes we propose in this section allow the structures to evolve.

A particular Complex Object cell interacts with a single Identity cell, so all the components of a complex object are assumed to reside in the same Identity cell. (A complex object may be replicated at another site as discussed in Section 4.6.)

The Complex Object layer adds a `type` field to all objects, as it hands them to the Identity Layer. The type field is used to record how the object is used by this layer. The Complex Object layer offers its clients an interface (not shown here) for accessing objects, analogous to that of the Identity Layer. For instance, the call `CO_Put(bag_of_bits)` is handled by adding the type `base` to the `bag_of_bits`, and calling `IL_Put(new_bag_of_bits)`. The `base` type indicates that this object is not one of the structural objects generated by the Complex Object layer.

4.5.1 Tuples

The basis for implementing any complex object is the *tuple* structure. A tuple is simply an object (of type `tuple`) containing an ordered list of object handles. The interface for tuples is:

- `CO_CreateTuple(list_of_handles):handle`:

  Creates a tuple containing the handles passed as parameters; returns the handle of the new tuple object.

- `CO_GetTuple(handle):list_of_handles`:

  Returns the list of handles in the given tuple.

Figure 4.3 illustrates two tuples. Tuple T_1 (created first) contains the handles of objects O_1 and O_2. We can represent this as $T_1 = \langle O_1, O_2 \rangle$. The second tuple T_2 is $\langle\langle O_1, O_2 \rangle, O_3 \rangle$. Note that one could also create the tuple $\langle O_1, O_2, O_3 \rangle$, but it is different from T_2.

4.5.2 Versions

Versions are a way of implementing updateable objects in an environment in which direct updates are not allowed. When using versions, we update an object by creating a "new" version of it. Versions support these functions:

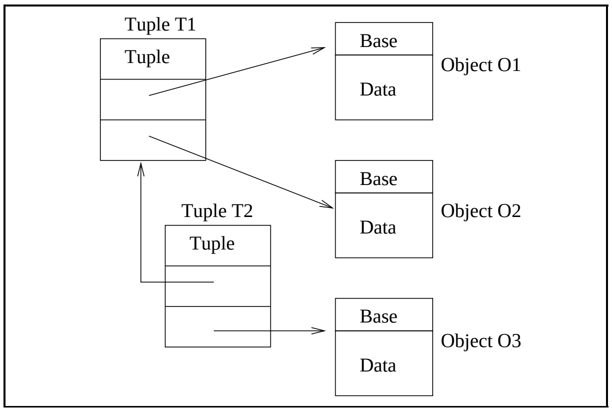

Figure 4.3: The tuple $<< O_1, O_2 >, O_3 >$

- `CO_CreateVersionObject(): handle :`

  Creates a new version object and returns its handle.

- `CO_Update( handle, new_version ):`

  Creates a new version of the object with the given handle.

- `CO_Read(handle):list_of_handles:`

  Returns the list of handles that are the current versions of the object.

- `CO_Versions(handle):list_of_handles:`

  Returns the list of all versions of the object.

Figure 4.4 illustrates how versions can be implemented using tuples. Object V_1 (type `version object`) is the "anchor" for the sequence of versions. Version 1 is recorded by the lower `tuple` object in the figure. Its list of handles contains (a) the handle of the anchor version object; (b) the handle of the object that constitutes this version; and (c) the handle for the previous version. (If this is the initial version, this last handle is null.) The upper `tuple` object records a second version. Because objects cannot be updated, the version "chain" goes from a more recent version to earlier ones. In addition, the anchor version object, which identifies this chain,

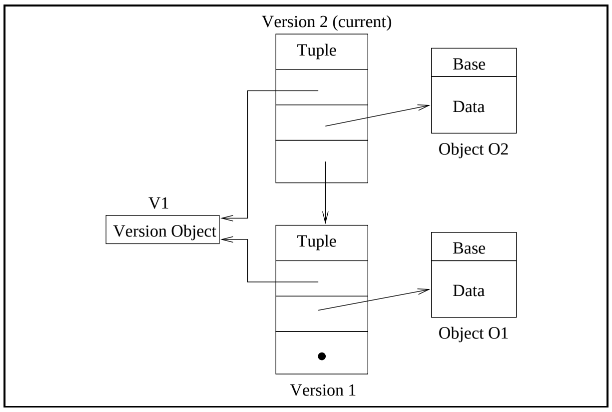

Figure 4.4: A document with versions v_1 and v_2

cannot contain a list of all versions. (We would need to update it as new versions are generated.) The structure of Figure 4.4 was created by the following sequence of calls:

- `CO_CreateVersionObject()`. This returns the anchor `V1`.
- `CO_Update( V1, O1 )`, where `O1` is the handle of the first version.
- `CO_Update( V1, O2 )`, where `O2` is the second version.

To read the latest version of `V1`, we use the call `CO_Read(V1)`, which returns a handle to `O2`. In our example there is only a single latest version, but as we discuss in Section 6, replicating a chain at several sites and independently updating it may lead to multiple latest versions.

The Update, Read, and Versions functions need to determine the latest version, given an anchor object `V`. This must be done indirectly. One way is to scan all `tuple` objects, looking for any that reference anchor `V`. The one(s) that are not referenced by other tuples are the latest versions. Another way is to build a disposable structure that maps anchors to their member objects. Such a structure can be built by scanning all `tuple` objects, and then incrementally maintained as new `CO_Update` calls are made. Our design ensures that this disposable structure is not essential for the long term survival of the AR.

To record that a version chain has "ended" (e.g., it is inaccessible), we can generate a new version that points to a distinguished `null` object. The `CO_Update` call will refuse to create new versions beyond this final one. (We could actually define several "ending" objects to indicate different semantics, e.g., the version chain is frozen, it should not be accessed.)

In summary, version objects provide a mechanism for "updating" and "deleting" AR information. Since this mechanism builds upon our immutable objects, it provides very reliable and long term storage.

4.5.3 Sets

Other structures can be implemented in a similar fashion. For example, Figure 4.5 illustrates how a set of objects can be implemented. Each member is a tuple that points to the set anchor (type `set`), and the actual member object. The interface for sets may include the functions:

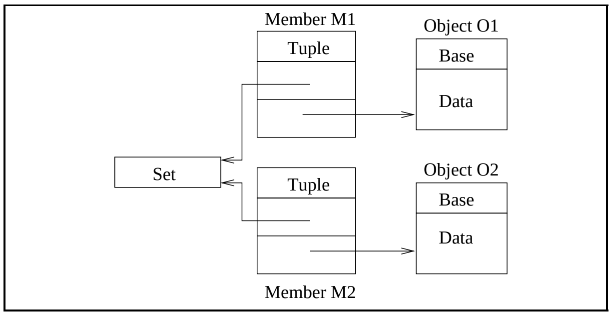

Figure 4.5: A set with two members

- `CO_CreateSet():handle`:

  Returns the handle of an empty set.

- `CO_InsertMember(set_handle, handle):`

  Inserts a member into a set.

- `CO_Member(set_handle, obj_handle):boolean`:

  Returns TRUE if the object `obj_handle` is a member of set `set_handle`.

We may have additional functions for sets such as Union, Intersection, and Difference, but these are not discussed here. As with versions, set membership can only be determined by scanning all objects, and looking for those with a given set anchor. Disposable structures can be implemented to make this process efficient. As we discuss in the next section, when sets are replicated at different sites, there may be temporary inconsistencies regarding membership.

4.6 Reliability Layer

The Reliability Layer copies objects from one site to another to increase the probability that objects persist for an extremely long time. This is achieved by establishing *replication agreements* between multiple sites to mutually maintain replicas of objects of a given *replication group*. For example, if the reliability layer at Site 1 establishes a replication agreement with Site 2 for objects of group G_1 (say a technical report series), then every time an object belonging to G_1 is created at one of the sites, a copy must be propagated to the other site. Agreements are multilateral: all members are responsible for backing up objects of the other members.

The Reliability layer adds two header fields to all objects, as it hands them to lower layers for storage. The `group` field records the replication group this object belongs to; i.e., it sets the desired level of replication. The group is selected by the client who creates the object in the first place. The second field, `agrmt`, is used to distinguish objects that represent agreements from those that do not.

Each replication agreement is recorded in a version complex object. The *agrmt* field in this object is set to True, and the *group* field is set to the identifier for this group. The content is a list identifying all the sites participating in the agreement. If the agreement changes, a new version is generated, with the new participants (and same `agrmt` and `group` fields). Note that *all* the objects that make up the version agreement for group G_1 are themselves in group G_1. Hence, they will also be backed up to participating sites. Moreover, the replication functions we describe here can be used to migrate a collection from one site X to another site Y (by first adding Y to a replication group, and then dropping X).

4.6.1 Reliability Interface

The interface of the Reliability Layer includes the following functions:

- `RL_NewAgreement(): gr_hdl`

  Creates a new replication agreement, identified by the returned `gr_hdl` handle. This handle is the group identifier, and should be given to all objects in the group.

- `RL_Participants(site_list, gr_hdl):`

  Makes `site_list` the current set of participants in `gr_hdl`.

- The interface also includes the functions in the Complex Object interface. For the functions that create objects, an additional parameter `gr_hdl` is added, to indicate the replication group to which they belong. Awareness functions are extended so that objects belonging to a given replication group can be requested.

4.6.2 Implementation

When a `RL_NewAgreement()` call is received, the Reliability cell simply calls `CO_CreateVersionObject()`, receiving a handle `G` that will be used as the group identifier. Next, the function `CO_Update(G, O1)` is called to create the initial version of the agreement. Object `O1` has its `agrmt` field set to True, its `group` field set to `G`, and its contents set to an empty set of sites. The result of the `RL_NewAgreement()` call is `G`, which can then be used by the client to create objects in this replication group.

An AR administrator then issues a `RL_Participants` call to record the participating sites. That call is issued at only one of the participating sites, since the site will immediately propagate the news to the other sites. The call generates a new version of the agreement (in the version chain anchored by `G`), containing the new list of participants.

Once an agreement is in place, the Reliability Layer can enforce it in a variety of ways. Here we illustrate one simple way, assuming Reliability cell A is the one actively ensuring Reliability cell B has copies for group `G`. (Cell B would perform a similar process concurrently.) Periodically, A requests from B its complete list of handles corresponding to object in group `G`. To comply, cell B uses its lower awareness services to obtain all object handles (in its storage partition), and forwards those in group `G` to A. Cell A performs a similar scan at its own site, and then compares the handles. If a handle is seen locally but not at B, that object must be copied to B. (Cell A asks cell B to create a new identical object. The object may have existed at B before, but it may have been corrupted.) Similarly, if an object is missing locally, it is requested from B and created at the local site.

When asked to replicate objects of a complex type, the reliability layer creates shallow duplicates. For example, let us suppose that a version object `V1` is created, together with a first version of a postscript technical report. Let us assume that all these objects are defined to be in group `G1`. Next, a second `V1` version is created (e.g., an updated report), but for some reason its group is defined to be `G2`. A site that is only in `G1` will receive the first version of the report, and not the second one. Thus, to ensure that a complex object is fully replicated, all of its components must be in the same group. Auxiliary tuple objects created by the Complex Object Layer do not have a replication group field, since they are generated implicitly by the Complex Object layer. However, these objects still need to be replicated, as part of the complex structure they participate in. To achieve their replication, we implicitly assume that the replication group of a tuple object is the union of the replication groups of the base objects it points to.

The stored replication agreement is used by a Reliability cell to "remember" its agreements in case of problems. Let us consider a few sample problems as an illustration. In our first scenario, Reliability cell A fails while participating in group `G`, and loses its state, but the latest agreement for `G` was not lost at the local site.

Cell A restarts by scanning the local site for all objects[1] with their `agrmt` field set, eventually finding the latest version of agreement `G`. From that point on, it resumes its backup work with the other participants. Any `G` objects lost during the failure will be reconstructed from the other participants.

In our second scenario, for example, when cell A recovers, no record of agreement G is found locally. Hence, cell A does not know it is participating in `G`. However, hopefully other `G` sites are active, and they will realize that A has lost objects, and will restore them. Since the agreement for `G` is in the group, it will also be restored.[2] Eventually A realizes there is an agreement in which it participates, and resumes its activity. (Cell A needs to periodically scan its local object to ensure it has accurate information.)

In our third scenario, the latest version of agreement `G` is lost, but some older version survives. When A recovers, it starts its activity with an out-of-date list of participants. This may cause it to temporarily miss some of the sites that contain replicas, and may cause it to send object copies to sites that are no longer participants. However, the latest version of agreement `G` will eventually make it to A, and A will eventually operate correctly. We emphasize that the only "damage" done in this scenario is the creation of non-needed replicas at sites that had dropped out of the agreement. While un-needed copies may waste some space, they in no way compromise the objects that are already stored.

The reliability layer guarantees an "epidemic" [20] propagation of copies. If we look at a given object `X` in group `G`, `X` will be, with extremely high probability, at all `G` sites. There may be periods of time when `X` is missing at some sites (e.g., a copy was corrupted), but it would take an unlikely sequence of failures to make it disappear from all `G` sites. Note that there is no notion of a distributed commit for `X`. Object `X` is committed when it is created at one site, and its probability of long term existence increases as copies are propagated. The fact that our objects are immutable,

[1] This assumes that Cell A knows what its local site is. We can agree in advance on fixed ports for the local layer interfaces.

[2] Object `G`, the anchor for the version chain, is not in group `G` since it was created before the group existed. However, the versions in `G` are in the group and are sufficient to reconstruct the latest version.

simplifies the protocol and increases the chances it will work correctly. In particular, there is no danger that the distributed `X` copies will become "inconsistent."

When a client creates an object `X`, the client may wish to know when it has been replicated at all `G` sites, so it knows it has reached its "extreme safety" mode. For this, we can add a function to the Reliability layer that checks if an object is found at all participating `G` sites.

When complex objects are in the same group, they get replicated and their copies converge. Sites may temporarily have incomplete information, but we do not view this as a strict inconsistency. For example, site A may think that a technical report is available in ASCII and Postscript, while site B may think it is available in ASCII and Word97. If this information is encoded as a set, eventually both sites will know about all three formats.

4.7 A Complete Example

In this section we give an example of how the layers described in the previous sections work together. In this example, we will have two repositories containing technical reports, one at Stanford and another one at MIT. These two sites have a replication agreement for all objects belonging to the technical report group TRG.

Let us suppose that an upper cell at Stanford wants to publish a technical report. The publisher anticipates that several versions of this document may be generated and decides to use a "Version" complex object. (For the sake of simplicity, we are assuming that each version of a technical report is just one object). First, the publisher asks the Reliability Cell at Stanford to create a new version object V belonging to the replication group TRG. Recall that the version object does not contain the data for the technical report (we will save this data as its first version). The Reliability Layer calls the Complex Object Layer function `CO_CreateVersionObject()`. In turn, the Complex Object cell generates the version object and saves it by calling the Identity cell, which calls the Object Store cell. As a result of these calls, the Reliability Layer obtains the handle of the version object V.

After creating the version object, the client is now ready to generate the first version of the technical report. First, the client creates the technical report object, TR_1, by calling the `Put()` function in the Reliability cell at Stanford. The reliability cell sets the group field to TRG and asks the lower layers to save the report. After creating TR_1, the client makes TR_1 a version of V by calling the `Update()` function in the Reliability Layer. The Reliability Layer passes the request on to the Complex Object Layer which generates an object, V_1, containing a pointer to V, TR_1, and the previous version (which is a NULL pointer in this case as this is the first version). At the left of Figure 4.6 we show the state of the Stanford site (at this moment, the MIT repository would be empty).

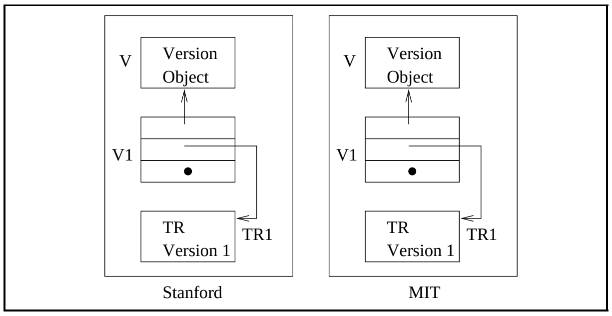

Figure 4.6: The repositories after replication.

As there is a replication agreement between MIT and Stanford for the objects in the Technical Report group, the MIT (or the Stanford) Reliability Cell will try sometime later to enforce the agreement by querying the other reliability cell and finding out that the newly created objects, TR_1, V, and V_1, are missing in the MIT site. As described in Section 4.6, the simple way of doing this query is to use the $IL\_Awareness()$ function to obtain all the handles in the other site and then compare those handles with the handles on our own site. A more efficient way of doing this query is to use the $IL\_Latest()$ function to discover which handles have been added to the repository since the last time it was visited. There are more efficient awareness algorithms that will be discussed in Chapter 5. After finding the handles of the missing objects, the replication process creates replicas of those objects in the MIT site. At this moment, the content of the repository is shown in Figure 4.6. (We do not show the Reliability Agreement Object that we are assuming was created earlier.)

Note that at this point we could have a synchronization problem if we concurrently add two new versions, one at MIT and the other at Stanford. Figure 4.7 illustrates this by showing the state after Stanford generated Version TR_2, and MIT independently created Version TR_3. When the replication process copies the new objects to the other sites, we end up with multiple latest versions, as shown in Figure 4.8. That is, the call `CO_Read(V)` will return both TR_2 and TR_3. We view this as an application "problem." Perhaps it was the intention to have multiple current versions for this report, i.e., the Stanford and MIT versions of a jointly authored paper. If this was not the intention, then the "report creation" layer should ensure that only one author at a time creates new versions of a report. This type of sequencing could be enforced by a synchronization service that is not discussed here.

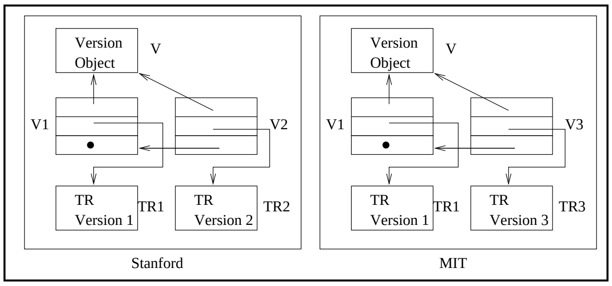

Figure 4.7: New versions at Stanford and MIT.

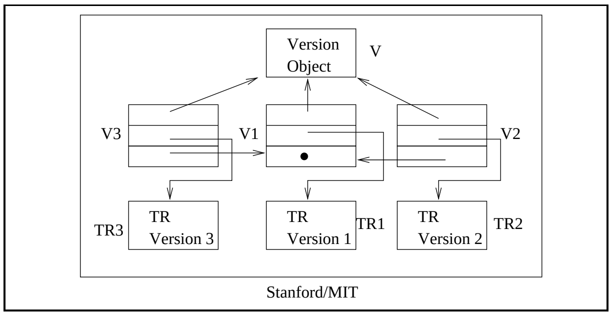

Figure 4.8: Inconsistent State.

Let us return to the state of the repository of Figure 4.6 and let us suppose that the Stanford Repository has a failure that completely destroys all its information. After this failure, the reliability process at Stanford cannot recover its data, since its Reliability Agreement Objects (that indicate where the replicas are) have been lost. However, some time later, the Reliability cell at MIT visits Stanford and it finds out that some objects, including the Replication Agreement Objects have been lost at Stanford. The Reliability Cell at MIT restores those objects (and potentially some others), allowing the Reliability cell at Stanford to also start recovering its destroyed digital objects.

4.8 Discussion

In this chapter we have studied an architecture for long-term archival storage of digital objects. We have argued that we can build a simple, yet powerful, archival repository by using signatures as object handles, not allowing deletions, having awareness services in all layers, and using only disposable auxiliary structures. We believe this architecture is well suited for a heterogeneous and evolving environment because each site only needs to agree on some very simple interfaces, on a signature computation function, and on some simple object header structure (e.g., for type and group fields). Although sites may use auxiliary structures, they need not agree on their details and use. There are no i-node tables, out-of-sync clocks, inconsistent indices that can cause us to lose or corrupt information. Since objects are never deleted or modified in-place, many sources of confusion are eliminated, yielding an extremely safe AR. Migration of information from an obsolete site to a new one is simple, and can be performed by the replication services.

4.9 Related Work

Several architectures have been proposed and implemented for digital libraries [3, 38]. These architectures focus on interoperability and distribution, but are not directly concerned with the problem of long-term reliability.

The task force on preserving digital information [15] has investigated the means of ensuring long-term safekeeping of information in digital archives. As in our architecture, the task force regards migration as an essential tool in preserving digital archives. However, the task force deliberately avoids defining the implementation details for a digital archive.

At the secondary device level, the Petal [48, 49] and Frangipani [81] projects have designed highly-available, scalable block-level storage systems that are easy to manage. The availability of the system is achieved by using data striping and redundancy. Although these projects consider the problem of long-term data reliability, their aim is a "file system" replacement. They allow in-place updates and deletions, and use application generated filenames (handles).

In the business world, Computer Output to Laser Disk (*COLD*) systems have been very successful in solving the problem of long-term archiving of data that is not frequently accessed. COLD systems were originally designed to replace microfiche and paper archival applications with online computer systems. A typical COLD system captures the output of a computer program and stores it. Typically, the storage media are CD-ROMs but nowadays other types of storage media (magnetic disks, RAID, magnetic tape, and re-writable laser disks) are also used [27]. COLD systems are monolithic with very few computers, all of them running exactly the same software. This is different from the heterogeneous environment we have considered. Storing data on a write-once COLD device forces the data to be immutable, as in our design. However, COLD systems always assume that some persistent storage is available on a write-many device, which can be used for some structures. We assume all AR storage is immutable.

Systems based on layering have proven effective especially in the area of networking. Specifically, the Open System Interconnection model (OSI) provides a standard that divides a network into seven layers with clear responsibilities [77].

Chapter 5

Archive Synchronization

5.1 Overview

In Chapter 4 we presented a general architecture for Archival Repositories. A key service that all layers in the architecture must provide is *object awareness*. Clients must be aware of changes at the repository, either new objects, deleted objects, or modified objects. The focus of this chapter is on awareness services. These are fundamental services which are necessary to decouple the storage of digital objects from their management (e.g., search, access rights, naming, payment). Furthermore, the traditional timestamp-based approach (used on the Web and in file system based stores), and other common schemes have some serious problems. Thus, in this chapter we carefully consider the spectrum of options for awareness services (even beyond the write-once paradigm) and suggest some solutions that have important strengths for digital libraries.

To illustrate some of the problems with "traditional" awareness schemes, consider our experience with the CS-TR Project [18]. Under this project, Computer Science departments and research laboratories (currently about 50 participating) provide access to their technical reports. For each report, a bibliographic record (file) is created, giving the basic meta-data for the report (e.g., title, authors, date). Various index services need to obtain new bibliographic records on a regular basis. One of these indexes was SIFT, a selective dissemination service at Stanford wherein users could "subscribe" to new reports by providing keywords that described their interests.[1] Each CSTR store provides an interface to obtain new records, created after some given timestamp.

First, we considered using as the timestamp of a bibliographic record the creation time of the file that held the record. With this scheme, a store could simply look for new files when a client requested new records. This option was ruled out because we wanted to be able to migrate and replicate records. For example, when the hardware holding a collection becomes obsolete, we need to move the records to a new file system (digital libraries are long lived, so we need to plan for this!), and this would generate new timestamps for unchanged records. Similarly, we wanted the replicas of

[1] SIFT is now operated by InReference; see `http://www.reference.com`. The CS-TR component was actually discontinued in 1996 when the service was commercialized.

a record at two stores to have the same timestamp, since otherwise an index service might consider one a new version of the other. For these reasons, we ruled out file system generated timestamps.

Instead, we used application generated timestamps: a field within the bibliographic record indicated when the record was created or modified by a librarian. Unfortunately, this scheme led to another problem: librarians would modify records (e.g., fix a typo in the author's name), and would forget to update the timestamp. Similarly, for one reason or another, records were created with old timestamps. All this caused services like SIFT to miss records and provide incorrect indexing.

Although we have assumed no *voluntary* deletions in our archival architecture, it is possible for files or documents to be *involuntarily* deleted (e.g., due to user error or media decay). Therefore, our synchronization algorithms need to be able to cope with deletions. Unfortunately, timestamps, either system or application generated, do not cope well with object deletion. That is, a store looking for objects with timestamps greater than a certain value, will not notice that some objects have disappeared entirely. This problem is of course clear to anyone who indexes Web sites. These sites are incapable of reporting deleted web pages, so we frequently see indexes that point to non-existent pages. While this may be acceptable for the Web, not all digital libraries will be satisfied with this type of awareness service.

An alternative to timestamp-based schemes that we carefully explore in this chapter are content-based schemes. In one simple version of these content-based schemes, the store can compute a signature (e.g., large checksum) for each object it holds. It sends this full collection of signatures to a client, which can then compare the signatures to the those for the objects it holds (or has indexed), yielding the identities of the changed objects. These schemes are especially attractive for digital libraries because they are not sensitive to how the bits are stored, only to the actual content of the object. However, it can be very inefficient to compute all the signatures and to send them to clients. Thus, in this chapter we also explore techniques to make content-based schemes practical.

In summary, this chapter makes two contributions:

- It surveys the spectrum of awareness options, highlighting the advantages and disadvantages of each. Interestingly enough, all the awareness options that we know of can be captured by a *single* algorithm with configurable options. We present this unifying algorithm, *UNI-AWARE*, and show how different choices for its parameters lead to different awareness mechanisms such as timestamp-based, log-based, triggered updates, signatures, and so on.

- For signature-based schemes, we present several enhancements that reduce the computation and communication costs involved. As stated above, we believe that these schemes have significant benefits in a digital library environment, in which the way objects are stored may vary from one site to another, and over time.

We start our discussion in Section 5.2 by defining the awareness problem more formally. Section 5.3 briefly surveys related work, and Section 5.4 surveys the design space for awareness mechanisms. Our unifying algorithm, UNI-AWARE, is presented in Section 5.5, while Section 5.6 discusses optimizations for the signature-based algorithms.

5.2 Problem Definition

A data store contains a set of digital objects $R = \{O_1, O_2, ...O_N\}$. Each object has a *handle*, $O_i.H$, that can be used by a client to retrieve the object from the store. In addition, each object may have other *attributes* such as:

- Timestamp, $O_i.TS$: a timestamp representing when the object was created or last modified.
- Store, $O_i.S$: a globally unique identifier for the store that contains the object.
- Name, $O_i.N$: a user readable name for the object, e.g., "STAN-CS-456-97." Names can often be used to retrieve objects, with the help of a name service that maps names into handles.
- Title, $O_i.TITLE$: the object title.

- Author, $O_i.AUTHOR$: the object's author(s).
- Body, $O_i.B$: the main body of the object, e.g., a postscript file for the object, or the full text in ASCII.

Various sets of attributes are used in practice, e.g., the Warwick Framework [46] or the Dublin Core [84]. Each defines a list of possible attributes and their types (e.g., strings, lists of strings, nested records). The attributes for an object (sometimes called the meta-data) can either be stored together with the object itself or in a linked but separate file, or they can be computed on the fly from the object. (For example, the title may be stored explicitly in a meta-data file, or it could be extracted by scanning the document for the string that follows "Title:.") For simplicity, in this chapter we assume that all attributes are simply part of each object O_i.

We model object removal and creation as deletions and insertions to set R. We model the modification of O_i (or any of its attributes) by the creation of a new object O_j. For modification, there are three cases to consider:

1. *Versions:* The old object O_i remains as a previous version. The handle of the new object, $O_j.H$, is necessarily different from $O_i.H$ in order to distinguish the two versions.
2. *In-Place Update:* The old object O_i is removed, and the handle of the new object is unchanged, i.e., $O_j.H = O_i.H$. This represents an "in-place" update of the object.
3. *Shadow Updates:* The old object is removed, and the new object has a new handle, i.e., $O_j.H \neq O_i.H$

A *client* to a data store needs to "learn" about all the digital objects in R that satisfy some filter predicate σ. The filter σ specifies what objects the client wishes to learn about. Learning about object O_i may mean receiving a complete copy of O_i and storing it locally (for reliability). Learning about O_i may mean receiving the digital object, extracting certain attributes of interest, adding those attribute values (together with $O_i.H$ or $O_i.N$) to an index, and discarding O_i. Learning may also mean simply receiving the attributes of interest, and not the full object. The client also has a "forget" operation for O_i that undoes the action performed when O_i was learned.

Given this framework, we can now define the goal of an awareness scheme: whenever a store creates a new object O_i that satisfies the filter of a client, that client must (at some not-too-distant future time) learn about O_i. Whenever a store removes an object O_i that satisfied the filter of a client, that client must forget O_i.

In closing this section, we note that in general, there are two types of handles: semantic and arbitrary. By looking at a semantic handle we may be able to infer something about one or more of the attributes of the identified digital object. For example, if handles are strings of the form `site.id`, then from $O_i.H$ we can determine $O_i.S$. An arbitrary handle conveys no information when examined by anyone other than the store that generated it. For the awareness problem, two types of semantic handles are of interest:

- *Sequential handles:* the lexicographical order of handles determines the order in which their digital objects were created. Thus, if $O_1.H < O_2.H$, we know that object O_1 was created (or last modified) before O_2. (Note that sequential handles cannot be used with in-place updates as defined above, since a modified object must keep its old handle.)

- *Content-sensitive handles:* Two digital objects O_1 and O_2 (and all their attributes) are identical if and only if their handles $O_1.H$ and $O_2.H$ are identical. In this case, we can detect duplicate objects simply by checking their handles. Similarly, if a store modifies an object in any way, its handle will change, making detection of the change easier. (Again, content-sensitive handles cannot be used with in-place updates.)

In Section 5.5 we discuss how sequential or content-sensitive handles can be used for awareness.

5.3 Related Work

The awareness problem is related to the problem of maintaining database replicas. Snapshots as a means of updating replicas were first introduced in [1]. The *Differential Refresh* Algorithm updates a previous snapshot by using a log [51]. Recent work in data warehousing improves the efficiency of these algorithms [45].

Remote file comparison algorithms are also related to the awareness problem. These algorithms attempt to locate disagreements between two or more large remote data files [55]. A class of algorithms that use randomized signatures to compare remote file copies is presented in [6]. The difference between these algorithms and our work is that they operate on a fixed set of pages, rather than with object deletion and creation. They also assume that few pages (objects) differ, while our algorithms do not make this assumption.

A related problem is the deployment of programs over a network. Current solutions use a repository that maintains the master copy of the application code. Remote nodes check the central node periodically for updates to the master copy. In case of an update, the remote node replaces its local copy with the updated application. Commercial software that solve this problem includes Marimba [54], NetDeploy [58], and BackWeb [5]. Although solutions to the awareness problem are solutions to the program deployment problem, there is a significant difference in scale. Program deployment solutions use file granularity and assume that the "master copy"contains a relatively small number of files.

5.4 The Client-Store Design Space

The performance and complexity of an awareness mechanism very much depends on how the store interacts with the client. In this section we briefly discuss the possible store-client interaction modes and parameters.

Push vs. Pull. Either the client or the store can be responsible for keeping the client up to date. With the *push model*, the data store is responsible for notifying the client when changes have occurred. With the *pull model*, it is the client's responsibility to query the data store for the changes. With a hybrid *hint-pull model*, the client is still responsible for requesting changes, but the store can independently provide a "hint" to the client that it is time to make a query.

A push model simplifies the design of the client since the client only needs to wait for changes. In addition, network traffic may be reduced since messages are sent only when there are indeed changes. On the negative side, with a push model, the store needs to know the identity of all clients, and must make sure that they receive the changes. A second problem was evident in the TULIP experiment, in which Elsevier Science distributed science and engineering journals to nine universities through the Internet [10]. The TULIP project initially used a push strategy, based on the assumption that all universities would be able to start receiving documents within the same time frame and that there would be enough storage space available at the universities to receive the information. However, these assumptions did not hold in practice and during the course of the project the strategy was changed to a pull model. A pull model, in contrast, has complementary characteristics: the store is simpler, and the client needs to do more work.

Stateful vs. stateless stores and clients. Data stores can be either stateful or stateless with respect to clients. A stateful store saves information concerning what objects clients have learned about. The information can be on a client per client basis, or it can be global. For instance, the store may remember the timestamp of the latest object seen by each client, or it may simply remember a single timestamp, that of the latest object seen by *all* clients. In the latter case, a client may receive duplicate information.

In a stateful store, the state can be persistent or volatile. If the state is volatile, then a failure can destroy the information, in which case clients will have to use some other awareness scheme. For example, if the store loses the latest timestamp information, clients will have to learn about all objects in the store (many not for the first time), instead of just the new ones.

If the store is stateless, then each client can save the state of its interaction with the store. For example, each client can remember the largest object timestamp it has seen. When it pulls updates from the store, it can ask for objects with a timestamp greater than this one.

Cognizant Clients and Sources. A cognizant store knows the identities of all its clients. It can use this knowledge to authorize pull requests from clients, or to decide what clients to push information to. A stateful store must be cognizant, but a stateless store need not be. An http server is an example of a non-cognizant, stateless store.

Although it may sound surprising, clients may or may not be cognizant of the stores they access. A Web crawler, for example, does not necessarily have a list of stores to access. Instead, it performs source discovery by examining the links in existing objects (e.g., URLs), and then by contacting the appropriate stores for the web pages.

Number of Clients per Data Store. The ratio of the number of stores to the number of clients is a parameter that strongly affects the performance of the various awareness schemes. If a store serves many clients, it may be hard for the store to keep state information on a client per client basis, forcing the use of a less effective awareness scheme. Also, the frequency of pushing or pulling may have to be reduced.

A client may have to interact with more than one store if, for instance, it is building an index over multiple collections. If the stores use different awareness schemes, then the client must be prepared to run the different protocols. The unifying algorithm we present next, UNI-AWARE, may actually be a convenient framework for such a client to coordinate the different interaction styles.

5.5 The UNI-AWARE Algorithm

In this section we present *UNI-AWARE*, a unified awareness algorithm that "covers" known schemes. Casting all schemes in this same framework makes it possible to clearly see their differences and strengths. In Section 5.6 we present an enhanced version of UNI-AWARE.

Figure 5.1 presents Algorithm UNI-AWARE. We stress that the pseudo-code in the figure is a logical explanation and not necessarily the best implementation of the algorithm. In the code, all functions prefixed with "Store" run on the data store, while functions prefixed with "Client" run on the client; however, these functions can be started remotely through remote procedure calls.

func Store_UniAware(**func** σ_{pre}, **func** σ_{post}, **func** Compress):
    $i_{obj} = \sigma_{pre}($ S); // S contains all data store objects
    c_{obj} = Compress(i_{obj});
    $vector_{store} = \sigma_{post}(c_{obj})$;
    return $vector_{store}$;

func Client_UniAware(**func** $Client\_\sigma_{pre}$, **func** $Client\_\sigma_{post}$,
    func $Store\_\sigma_{pre}$, **func** $Store\_\sigma_{post}$,
    func Compress, **func** Decompress):
    $i_{obj} = Client\_\sigma_{pre}($ C); //C has all objects the client has learned about
    c_{obj} = Compress(i_{obj});
    $vector_{client} = Client\_\sigma_{post}(c_{obj})$;
    $vector_{store}$ = Store_UniAware($Store\_\sigma_{pre}$, $Store\_\sigma_{post}$, Compress);
    Learnobj= $vector_{store}$ - $vector_{client}$;
    ForgetObj = $vector_{client}$ - $vector_{store}$;
    NewObjs = Decompress(LearnObj);
    Learn NewObjs;
    OldObjs = Decompress(ForgetObj);
    Forget OldObjs;

Figure 5.1: UNI-AWARE Algorithm

Algorithm UNI-AWARE uses six different *Custom Functions*; an instantiation of these functions tailors UNI-AWARE to execute a particular awareness *instance* (e.g., timestamps, signatures). A custom function takes as input a list of elements, that we call a *vector*, performs some operation on those elements and constructs another vector. We will now discuss the six functions.

The $Compress()$ function receives a vector of digital objects and returns a vector of "compressed" objects. Intuitively, the compressed version of an object is a summary containing all that is necessary for detecting changes under a particular scheme. Two examples of valid compression are: a function that returns the LZW compression of a object [47], and a function that returns a pair $< object\_handle, update\_time >$. After a change is detected, the client may need to access the changed or new object (e.g., to make a replica). For this it uses the $Decompress()$ function, an inverse of Compress. This function receives a vector of compressed objects and returns a vector of objects or object handles. In our example above, the first decompression function would apply the LZW decompression algorithm, which will return the object itself. In the second example, Decompress would be a function that returns the object handle.

The store pre-selection function, $Store\_\sigma_{pre}()$, is used to eliminate from consideration objects in which the client has no interest. For example, in the CS-TR environment, a client may indicate interest only in objects containing the word "database." The store post-selection function $Store\_\sigma_{post}$ is used to eliminate from consideration objects the client has already learned about, e.g., those with a timestamp greater than some value.

In the UNI-AWARE Algorithm, we conclude that an object needs to be forgotten when we do not receive its compressed version from the data store that sent the object originally. Since in this case the store did not apply its pre and post filtering function on this missing object, the client must check if it was interested in the original object (otherwise, it did not learn about it, so there would be no need to forget it). For this purpose it uses functions $Client\_\sigma_{pre}()$ and $Client\_\sigma_{post}()$, which are analogous to those on the store side.

A cycle in the awareness process starts with a call to $Client\_UniAware()$. If we are using the pull model, the client makes this initial call; if we we are using a push model, the store remotely invokes this call at the client. (With the push model, the call can be initiated periodically, or can be triggered by an update to the store.) First, the $Client\_UniAware()$ function generates a vector of compressed objects, $vector_{client}$, representing those objects the client has learned about in the past. (Keep in mind that this is a conceptual description; for instance, the client may simply cache $vector_{client}$ so it does not have to recompute it.)

Next the client will ask the data store to generate, in a similar fashion, the $vector_{store}$. This vector is sent to the client which compares it to its own. The objects represented by $vector_{store}$ but not in $vector_{client}$ are the ones the client must learn about. Those represented in $vector_{client}$ but not in $vector_{store}$ are the ones the client must forget. Finally, the client decompresses the two difference sets to obtain handles for the objects that need to be learned or forgotten.

In Figure 5.2 we present a simple UNI-AWARE example. The objects at the store are shown in the top left. Let us also assume the client has built an object cache, and its contents are shown in the top right. Let us assume that the client is interested in all objects ($Store\_\sigma_{pre}()$ and $Client\_\sigma_{pre}()$ always evaluate to true), and that it is not possible to know from compressed objects if they have been learned by the client ($Store\_\sigma_{post}$ and $Client\_\sigma_{post}$ always return true). When UNI-AWARE runs, the $vector_{client}$ and $vector_{store}$ vectors are computed as shown in the figure, assuming that an object is compressed into its handle and some attribute C. When $vector_{store}$ is sent to the client, the client finds the handles of the inserted objects by performing $vector_{store} - vector_{client}$, and the handle of the deleted objects from $vector_{client} - vector_{store}$. The update of the object with title "Paper2" generates a forget operation for the old version followed by a learn operation of the new version. (We can optimize the algorithm to detect these deletion-insertion actions on the same object, in order to do a single modify operation. However, we do not discuss this in this chapter.)

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2'
TR4	Paper 4

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3

$vector_{store}$: [TR1 | C1] [TR2 | C2'] [TR4 | C4]

$vector_{client}$: [TR1 | C1] [TR2 | C2] [TR3 | C3]

$LearnObj = vector_{store} - vector_{client}$: [TR2 | C2'] [TR4 | C4]

$ForgetObj = vector_{client} - vector_{store}$: [TR2 | C2] [TR3 | C3]

Figure 5.2: Example of the UNI-AWARE Algorithm

In the following subsections we explore different alternatives for the Custom Functions. By changing the compression, decompression and post-selection functions we will produce four families of strategies: snapshot algorithms, timestamp algorithms, log-based algorithms, and signature algorithms. The pre-selection functions do not change in these instances. They are used only to support client filtering, and will not be included in our discussion.

5.5.1 Snapshot Algorithms

The Snapshot Algorithms use the simplest definitions of the Custom Functions. They only require that a store produce a snapshot, i.e., a vector of all objects at the store. We present two instances in this family.

The Simple Snapshot Algorithm uses the Customs Functions in Figure 5.3. These Custom Functions simply return their argument without change. This implies that the store sends all of its objects to the client (whether they changed or not) in order to update it. This is clearly wasteful, but on the positive side, the algorithm requires very little functionality from the store, and could be useful for small stores.

func Compress (obj_{vector}):	**func** Decompress (obj_{vector}):
return obj_{vector};	return obj_{vector};
func $Store\_\sigma_{post}$(obj_{vector}):	**func** $Client\_\sigma_{post}$(obj_{vector}):
return obj_{vector};	return obj_{vector};

Figure 5.3: Custom Functions for the Simple Snapshot Algorithm

The *Handle Snapshot Algorithm* only works with versions or shadow updates (no in-place updates), and when handles of deleted objects are not reused. (Content-sensitive handles are a special case of this scenario.) In this case, the snapshot need only include the handles of all store objects. When compared with the previous snapshot, a new handle in the current snapshot will indicate an insertion, while a missing handle will indicate a deletion. The Customs Functions for this algorithm are given in Figure 5.4. (Recall that $O.H$ is the handle of object O.)

```
func Compress ( obj_vector ):          func Decompress ( obj_vector ):
    result = [ ];                          return obj_vector;
    for obj in obj_vector:             func Client_σ_post( obj_vector ):
        result += obj.H;                   return obj_vector;
    return result;
func Store_σ_post( obj_vector ):
    return obj_vector;
```

Figure 5.4: Custom Functions for the Handle Snapshot Algorithm

The Handle Snapshot Algorithm can be extended to include other object attributes in the compressed vector, in addition to just the handle. The additional attributes can then help us detect modifications, in case the handle is not sufficient. This variation is actually the Handle-Signature Algorithm described later on.

5.5.2 Timestamps

If timestamps at the store can be used to order objects by their insertion or modification time, then we can reduce the number of objects sent to the client. The first instance that uses this idea is the *Timestamp Algorithm*, whose functions appear in Figure 5.5. This instance only works with *no-delete* stores. A store is no-delete if it never removes objects. Modifications are allowed, either with versions or in-place. (Shadow modifications are not allowed since the old versions cannot be removed.)

In the Timestamp Algorithm, the store only includes objects with a timestamp greater than T_m, a marker timestamp. This marker was computed in the previous invocation of the algorithm, and tells us what objects the client has already learned about. The code in Figure 5.5 computes the maker timestamp for the next invocation, T'_m, in two places. This is not really needed: the client can compute T'_m and pass it to the store as part of function $Store\_\sigma_{post}$, or the store can compute it and save it for the next iteration.

In Figure 5.6 we present a sample execution of the Timestamp Algorithm. In

func Compress (obj_{vector}):
  return obj_{vector}
func $Store\_\sigma_{post}$(obj_{vector}):
  T'_m = max timestamp in obj_{vector}
  return objects O with $O.TS > T_m$

func Decompress (obj_{vector}):
  T'_m = max timestamp in obj_{vector}
  return obj_{vector};
func $Client\_\sigma_{post}$(obj_{vector}):
  return [ ];

Figure 5.5: Custom Functions for Timestamp Algorithm

this figure, we can see that the client has learned about all the objects except the one with handle TR3. Thus, the marker computed in the last invocation is T2. The algorithm starts with the store creating $vector_{store}$ with all objects with timestamps greater than T2. The client then compares this vector to its own version to discover that it is missing TR3. In the process, the next marker is computed to be T3.

Data Store

Handle	Title	Timestamp
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3

Client

Handle	Title	Timestamp
TR1	Paper 1	T1
TR2	Paper 2	T2

Marker = T2

$vector_{store}$:

TR3	Paper3	T3

$vector_{client}$: $empty$

$LearnObj = vector_{store} - vector_{client}$:

TR3	Paper3	T3

$ForgetObj = vector_{client} - vector_{store}$: $empty$

Next Updated Marker = T3

Figure 5.6: Example of the Timestamp Algorithm

A variation of the Timestamp Algorithm in which we do not ship full objects is the *Handle-Timestamp Algorithm.* (We do not show its Custom Functions.) Here the Compress function extracts the handle and the timestamp of each object. (If handles are sequential, see Section 5.2, handles can also play the role of timestamps) The timestamps are used as in the Timestamp Algorithm to eliminate objects of which the client is already aware. When the client compares $vector_{store}$ and $vector_{client}$, it compares both the handles and timestamps. The benefit with respect to the Handle Snapshot Algorithm, is that now we can cope with in-place modifications or handle reuse, because the timestamp lets us detect modifications when the handle does not change. In summary, the Handle-Timestamp Algorithm works with no-delete stores and either version or in-place updates.

The reason we restrict the Timestamp and the Handle-Timestamp Algorithms to no-delete stores is that they cannot handle deletions. The problem with deletions is that after an object is removed, the data store does not keep any record of it. Therefore, the data store cannot tell the client to delete it.

There are two approaches for handling deletions. In the first approach, the client performs *lazy deletions*. That is, the client keeps all deleted entries until an error occurs, or until the client runs an algorithm, such as the Snapshot Algorithm, that can handle deletions. An error occurs, for example, if a user searches the client index to get the handle for an object of interest, and then goes to the store to look for that deleted object. In such an event, the user can ask the client to delete the index entry for the deleted object.

The second approach is for the data store to keep *tombstones* for deleted objects. When using tombstones, for each deleted object D_i, the store inserts a tombstone object D'_i with the same handle as D_i ($D'_i.H = D_i.H$), and an attribute indicating that D'_i is a tombstone. The timestamp of D'_i is its creation time (i.e., D_i's deletion time). The only change to the Timestamp Algorithm (or to the Handle-Timestamp Algorithm) is that when the client learns of the creation of tombstone object D'_i, it forgets D_i.

The disadvantage of using tombstones is that they take up space in the data store. However, tombstone space D'_i can be reclaimed when the store knows that all clients have learned of D'_i. This occurs when all client marker timestamps are greater than $D'_i.TS$. (This requires knowledge of all clients by the store.) Alternatively, the data store can unilaterally discard all tombstones older than a certain time marker T_{cut}. If a client then uses a marker T_m smaller than T_{cut}, the request fails and the client must use a simple algorithm such as the Snapshot Algorithm.

5.5.3 Logs

A common metadata structure kept by stores (mainly for crash recovery) is a log. The log is a sequence of records that registers all the update activities in the database [75].[2] Each log record is conceptually of the form $< LSN, H, A, B >$, where LSN is a unique sequential number, H is the handle of the affected object, A is the action performed (e.g. update, deletion, insertion), and B is some information about the new object body (such as a delta, the new object body).

There are essentially two ways to use logs. The simplest is to use the LSNs as timestamps for objects. That is, for each object with handle $O.H$, we can use the largest LSN in log records that contain $O.H$ as its "timestamp." For example, say that the record with LSN = 21 registers the creation of object $O_1.H$, that record with LSN = 35 registers its modification, and there are no other records with $O_1.H$. Then the number 35 is O_1's current timestamp. The Timestamp or the Handle-Timestamp Algorithms can be used directly under this interpretation of timestamps.

A second way to use logs is to have the store report changes by shipping the log records that describe the changes. This gives us the Log Algorithm, whose Custom Functions are given in Figure 5.7. A marker log sequence number L_m (computed in the previous invocation) tells us what records need to be shipped (analogous to T_m in the Timestamp Algorithm). When the client gets a new NewObjs vector to learn, it replays the log actions (in LSN order) and takes appropriate actions on its local state. For example, if a log record says object O_1 was deleted, then the client forgets O_1. If the record says some text was inserted into O_2, then the client may index the new text.

The Log Algorithm requires substantial sophistication on the part of the client, since it needs to understand the store actions. Thus, the algorithm may only be applicable in limited situations, but in those cases it can effectively reduce the amount of data that must be sent to the client. As with tombstones in the Timestamp Algorithm, there is also the problem of limiting the size of the log. Similar techniques

[2] For our purpose, we will consider a log at the level of objects. If the system provides a lower granularity log, we can simulate an object-level log by coalescing several log records. Similarly, we will ignore log records that are not related to updates in objects.

```
func Compress ( obj_vector ):
    result = [ ];
    for obj in obj_vector:
        result + = [obj.H, obj.H log records];
    return result;
func Store_σ_post( obj_vector ):
    discard all log records LSN ≤ L_m
    if object has no log records left
        remove object from obj_vector
    L'_m = max LSN in obj_vector
    return the new obj_vector;

func Decompress ( obj_vector ):
    L'_m = max LSN in obj_vector
    return obj_vector;
func Client_σ_post( obj_vector ):
    return [ ];
```

Figure 5.7: Custom Functions for the Log Algorithm

to those described in Section 5.5.2 can be used.

5.5.4 Signature Algorithm

The Timestamp and Log Algorithms use metadata (timestamps, log records) to reduce the amount of data sent to the client. As discussed in the Introduction, it is difficult to consistently manage timestamps (or logs) as stores fail or objects migrate from one computer to another. An algorithm like the Simple Snapshot does not rely on metadata for change detection, and is hence much more robust in the face of operating system changes or failures. In this section we enhance that algorithm so that all full objects need not be sent to the client.

The idea is to send, for each store object, its handle and a *signature* that "summarizes" its content. A signature is a token that (i) has a high probability of being unique for each object in the data store, and (ii) changes when the content of the object changes. Several functions comply with these requirements including Cyclic Redundancy Checks (CRC) and checksums. The probability that a signature is unique for a given object content can be made arbitrarily low by using more bits in the signature. With say 128 bits, the probability that two objects will have the same signature is insignificant, so they can be used safely for detecting changes (unless one is the type of person that worries about our Sun going supernova). Signatures do not have to be recomputed each time they are needed; instead they can be cached. However, the cache need not be reliable, since if the signatures are lost or corrupted, we can always re-generate them by re-applying the signature algorithm to all the objects. Incidentally, if handles are content-sensitive (see Section 5.2), then we do not need signatures, as handles can play the role of signatures.

Timestamps also have the two properties mentioned in the previous paragraph, so they can be used as signatures (assuming the system generates unique timestamps as each object is created or modified). However, with timestamps we do need system support since they cannot be inferred from the content, so in this sense they are less robust than content-based signatures.

The Custom Functions for the Simple Signature Algorithm are presented in Figure 5.8. The compression function produces a vector with tuples containing the object handle and signature. This vector is sent to the client, who compares it to its own vector to detect changes. For new or modified objects, the client uses the handle to request the object so it can learn about it. In Figure 5.9 we present an example for the Simple Vector Algorithm.

```
func Compress ( obj_vector ):
    result = [ ];
    for obj in obj_vector:
        result + = [obj.H,signature(obj)];
    return result;
func Store_σ_post( obj_vector ):
    return obj_vector;

func Decompress ( obj_vector ):
    result = [ ];
    for obj in obj_vector:
        result + = obj.H;
    return result;
func Client_σ_post( obj_vector ):
    return obj_vector;
```

Figure 5.8: Custom Functions for the Signature Algorithm

Even though the Simple Vector Algorithm is more efficient than the Simple Snapshot Algorithm, it still must send many signatures to the client, one for each object in the store. In Section 5.6 we discuss improvements to reduce this cost.

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2'
TR4	Paper 4

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3

$vector_{store}$:

TR1	signature(Paper1)
TR2	signature(Paper2')
TR4	signature(Paper4)

$vector_{client}$:

TR1	signature(Paper1)
TR2	signature(Paper2)
TR3	signature(Paper3)

$LearnObj = vector_{store} - vector_{client}$:

TR2	signature(Paper2')
TR4	signature(Paper4)

$ForgetObj = vector_{client} - vector_{store}$:

TR2	signature(Paper2)
TR3	signature(Paper3)

Figure 5.9: Example of the Signature Algorithm

5.5.5 UNI-AWARE Summary

The following table summarizes the UNI-AWARE options we have discussed.

Algorithm	Bytes Transmitted	Need Metadata	Limitations
SimpleSnapshot	$R + DN$	No	None
Handle Snapshot	$R + LN$	No	No in-place updates; no handle reuse
Timestamp	$R + DU$	Yes	no-delete store; no shadow updates
Handle-Timestamp	$R + 2LU$	Yes	no-delete store; no shadow updates
Handle-Timestamp with Tombstones	$R + 2LU$	Yes	None
Log	$R + 2LU$	Yes	None
Signature	$R + 2LN$	No	May miss updates (arbitrarily low probability)

The "Bytes Transmitted" column gives a *very rough* estimate of the communication cost per invocation. For this, we assume that the client must first send R bytes to the store, that there are a total of N objects, each of size D, that the client is interested in, and that of these, U have been inserted, deleted or updated since the previous invocation. We also assume that timestamps, LSNs, log records and signatures are all L bytes long. The two other columns in the table indicate if the algorithm requires system-supported metadata, and if there are any logical (i.e., not performance related) limitations.

5.6 Distributed UNI-AWARE Algorithm

The Signature Algorithm is desirable because it does not use system meta-data such as timestamps or logs. However, it does send substantial data to the client, even if few changes occurred. In this section we present a distributed algorithm, DIST-UNI-AWARE, that reduces this cost. We also present this algorithm as a unified algorithm that can be tailored to a variety of specific instances.

Intuitively, DIST-UNI-AWARE starts by sending a single "compressed" version of the objects in the store (as opposed to a compressed version of each object). If this summary (e.g., signature) matches what the client has, then the client knows immediately that there were no changes and no further messages are needed. If there were changes, the client asks the store to split the objects into groups and to send it a compressed summary of each group. The process then repeats itself by having the client check if the objects in each group have changed. Figure 5.10 presents Algorithm DIST-UNI-AWARE.

In order to form groups of objects, both the store and the client agree on a conceptual ordering of the objects. For example, the objects can be ordered by their handle or their title. Given this ordering, the store can describe a particular group (subset) by the minimum and maximum objects according to the ordering. In addition, since the store usually wishes to describe multiple groups, it is convenient to include in each group descriptor the maximum object of the previous group, and the minimum object of the next group. Thus, in algorithm DIST-UNI-AWARE the

```
func Store_DistUniAware( ... custom functions ... ):
    i_obj = σ_pre( S ); // S contains all objects in the data store
    c_obj.GroupInfo.Min = min(i_obj.H);
    c_obj.GroupInfo.Max = max(i_obj.H);
    c_obj.GroupInfo.NextMin = NULL;
    c_obj.GroupInfo.PrevMax = NULL;
    c_obj.Data = Compress( i_obj );
    vector_store = [σ_post( c_obj )];
    return vector_store;

func Store_VectorAs( GI, ... custom functions ... ):
    i_obj = σ_pre( S ); // S contains all objects in the data store
    vector_store = Group(i_obj, GI);
    return vector_store;

func Client_DistUniAware( ... custom functions ... ):
    vector_store = Store_DistUniAware( ... custom functions ... );
    Client_ProcessVector( vector_store, ... all custom functions ... );

func Client_VectorAs( GI ):
    if GI.Min not in σ_pre(C): // C contains all objects in the client
        Learn GI.Min;
    if GI.Max not in σ_pre(C):
        Learn GI.Max;
    Forget handles in σ_pre(C) inside interval (GI.PrevMax...GI.Min);
    Forget handles in σ_pre(C) inside interval (GI.Max...GI.NextMin);
    i_obj = objects in σ_pre(C) with handles in interval [GI.Min...GI.Max];
    c_obj.Data = Compress( i_obj );
    c_obj.GroupInfo = GI;
    vector_client = σ_post( c_obj );
    return vector_client;

func Client_ProcessVector( vector_store, ... custom functions ... ):
    for elem_store in vector_store:
        vector_client = Client_VectorAs( elem_store.GroupInfo );
        if (elem_store.Data ≠ vector_client.Data):
            if elem_store.GroupInfo.Min == elem_store.GroupInfo.Max:
                Learn elem_store.GroupInfo.Min;
            else:
                v_store=Store_VectorAs(elem_store.GroupInfo, ... custom functions ... );
                Client_ProcessVector(v_store, ... custom functions ... );
```

Figure 5.10: DIST-UNI-AWARE Algorithm

store will send to the client a vector of group descriptors, VG. Each descriptor $VG[i]$ applies to a group of store objects and contains two attributes: $VG[i].Data$, the compressed objects (e.g., their combined signature), and $VG[i].GroupInfo$, the bounds record. A $VG[i].GroupInfo = I$ record contains in turn $I.Min$, an identification (e.g., handle) of the minimum object in group $VG[i]$; $I.Max$, an identification for the maximum object; $I.PrevMax$, the maximum of the previous group; and $I.NextMin$, the minimum of the next group.

Algorithm DIST-UNI-AWARE uses the following Custom Functions. The preselection functions work as before, except that for simplicity we now have a single function σ_{pre} used at both the store and the client. The $Compress()$ function receives a vector of digital objects and returns a single "compressed" data item that represents all the objects in the vector. There is no Decompress function; instead there is *Group*() function that packages a set of objects into groups of objects. Finally, the post-selection function, σ_{post} operates on group descriptors (as opposed to a vector of objects).

A cycle in the awareness process starts with a call to $Client\_DistUniAware$. The client requests to the data store a group vector by calling the function $Store\_DistUniAware()$. The data store produces and sends to the client an initial group vector that spans all of its objects. Then, the client generates a vector equivalent to the one sent by the data store. An equivalent vector is a vector that has the same grouping information (i.e. boundaries) as another. When generating an equivalent vector, the client may need to learn about an object when it does not have one of the group boundaries, or when forget objects that are outside the boundaries of the group and the boundaries of the adjacent groups. The client's equivalent vector is compared with the vector sent by the data store. If their Data attributes are equal, the client concludes that the group of documents has not changed, and no more processing is needed. On the other hand, if they are different, the client asks the data store to split the group vector into multiple vectors, and proceeds recursively with the same algorithm. The recursion stops when groups have only one object. At this stage, the client can decide if it needs to learn about the object by comparing the data store compressed version of the object with its own compressed version.

5.6.1 Hierarchical Signatures

In this section we describe a DIST-UNI-AWARE instance, the *Hierarchical Signatures* Algorithm, that uses signatures of objects in a group.

func Group(obj_{vector}, GI):
  s_{handle} = sorted list of handles of obj_{vector};
  delete from s_{handle} all handles $<$ GI.Min;
  delete from s_{handle} all handles $>$ GI.Max;
  // split divides list s_{handle} in two equal parts
  (s1, s2) = split(s_{handle});
  $v_{low}.GroupInfo.PrevMax$ = GI.PrevMax;
  $v_{low}.GroupInfo.Min$ = GI.Min;
  $v_{low}.GroupInfo.Max$ = max(s1);
  $v_{low}.GroupInfo.NextMin$ = min(s2);
  low_{data}=objects of obj_{vector} with handles in s1;
  $v_{low}.GroupInfo.Data$=Compress(low_{data});
  $v_{high}.GroupInfo.PrevMax$ = max(s1);
  $v_{high}.GroupInfo.Min$ = min(s2);
  $v_{high}.GroupInfo.Max$ = GI.Max;
  $v_{high}.GroupInfo.NextMin$ = GI.NextMin;
  $high_{data}$=objects of obj_{vector} with handles in s2;
  $v_{high}.GroupInfo.Data$=Compress($high_{data}$);
  return $[v_{low}, v_{high}]$;
func Compress (obj_{vector}):
  return signature(obj_{vector});
func $Client\_\sigma_{post}$($group_{vector}$):
  return $group_{vector}$;
func $Store\_\sigma_{post}$($group_{vector}$):
  return $group_{vector}$;

Figure 5.11: Custom Functions for the Hierarchical Signature Algorithm

The algorithm uses the Custom Functions of Figure 5.11. The compression function produces a single signature from a vector of digital objects. The grouping function receives a set of objects and information about a group, and produces two vector groups each with half the content of the original group. Digital objects are grouped together based on the value of their handles.

Figure 5.12 presents an execution example of the Hierarchical Signatures Algorithm. In this example, the client has learned about all the digital objects in the data store, with the exception of the object with handle TR9 that was just inserted, and TR6 which was updated; the client also needs to forget the object with handle TR5 that was deleted. In the first iteration of the algorithm, the data store groups all digital objects in a single group, and sends a vector containing the group information and the signature of $< Paper1...Paper9 >$ to the client. This group information is shown as the first four boxes in $vector_{store}$ under Round 1. It includes the handle of the last object in the previous group (null because there is no previous group in this case), the handle of the first object in the group (TR1), the handle of the last object in the group (TR9), and the handle of the first object in the next group (also null).

Next, the client uses the received group information to build the equivalent vector, $vector_{client}$, with all digital objects it has learned about with handles between TR1 and TR9. As the client does not have the object with handle TR9, it can immediately conclude that it needs to be learned. Assuming the client does so, it can then construct the equivalent vector, and notices that the signature in it (the data component) differs. Thus, the client starts Round 2 by requesting the data store to split the initial group. In Round 2, the data store forms two group vectors and sends them to the client. With the grouping information of these vectors, the client can immediately conclude that TR5 needs to be forgotten. (This is because TR5 is in between the maximum handle of the first vector and the minimum handle of the second vector.) The client forgets about TR5 and builds the two equivalent vectors. This time, one of the group vectors is the same as the vector sent by the store and no more processing is needed. However, as the other vector is different, the client starts Round 3 by requesting the data store to subdivide it. In Round 3, the store sends information on group TR6, TR7, and on group TR8, TR9. The second group matches what the client has, but the store is asked to further split the first group. The store does so, and in Round 4 sends individual signatures for TR6 and TR7. When the client receives the information on these atomic groups, it proceeds as with UNI-AWARE and concludes that it needs to forget the old version of TR6 and learn about the new version of TR6.

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3
TR4	Paper 4
TR6	Paper 6'
TR7	Paper 7
TR8	Paper 8
TR9	Paper 9

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3
TR4	Paper 4
TR5	Paper 5
TR6	Paper 6
TR7	Paper 7
TR8	Paper 8

Round 1:

Vectors from Store:

NULL	TR1	TR9	NULL	s(Paper1..Paper9)

Equivalent Vectors in Client:

NULL	TR1	TR9	NULL	s(Paper1..Paper9)

Client concludes:

LEARN TR9 and start round 2 as s(Paper1..Paper9) ≠ s(Paper1..Paper9)

Round 2:

Vectors from Store:

NULL	TR1	TR4	TR6	s(Paper1..Paper4)
TR4	TR6	TR9	NULL	s(Paper6'..Paper9)

Equivalent Vectors in Client:

NULL	TR1	TR4	TR6	s(Paper1..Paper4)
TR4	TR6	TR9	NULL	s(Paper6..Paper9)

Client concludes: FORGET TR5 and start round 3 with the second vector as s(Paper6'..Paper9) ≠s(Paper6..Paper9)

Round 3:

Vectors from Store:

TR4	TR6	TR7	TR8	s(Paper6',Paper7)
TR7	TR8	TR9	NULL	s(Paper8..Paper9)

Equivalent Vectors in Client:

TR4	TR6	TR7	TR8	s(Paper6,Paper7)
TR7	TR8	TR9	NULL	s(Paper8..Paper9)

Client concludes: Start round 4 with the first vector as s(Paper6',Paper7) ≠ s(Paper6,Paper7).

Round 4:

Vectors from Store:

TR4	TR6	TR6	TR7	s(Paper6')
TR6	TR7	TR7	TR8	s(Paper7)

Equivalent Vectors in Client:

TR4	TR6	TR6	TR7	s(Paper6)
TR6	TR7	TR7	TR8	s(Paper7)

Client concludes: LEARN TR6

Figure 5.12: Example of the Hierarchical Signature Algorithm

5.6.2 Hierarchical Timestamps

The *Hierarchical Timestamp Algorithm* is similar to the Hierarchical Signature Algorithm, except that the "signature" of a group of objects is now the largest timestamp of the group. If none of the objects in the group has been modified, then the group timestamp (i.e., the maximum timestamp) will match the group timestamp stored at the client. Otherwise the client will request that the group be split. We do not show the Custom Functions for this algorithm since they are very similar to those of the Hierarchical Signature Algorithm.

Timestamps are more effective than signatures for detecting changes because in that case the timestamps of different objects will never match. Signatures of different objects could match, but the probability can be made arbitrarily small by making the signatures large (e.g., 128 bits). Of course, as we have discussed, timestamps depend on the correct operation of the underlying operating or file system.

5.6.3 Performance Issues

The performance of the DIST-UNI-AWARE Algorithms depends very much on the number of new or changed objects between invocations. If there are no changes, only one round is required. If there is a single change, it takes roughly $\log_2 N$ rounds, where N is the number of digital objects. If there are two changes, we still require $\log_2 N$ rounds, but we may need to send twice as much data. As the number of changes increases, we end up having to split more and more groups. Eventually, with enough changes, every single group must be split, and the cost is much higher than if we had just sent all object signatures (or timestamps) outright.

There are many enhancements that can improve the performance of the algorithm as we have presented it. For example, groups can be split by factors of f instead of 2, reducing the number of rounds to $\log_f N$. This means that the store will send information for f groups at one time, but the cost of this may not be much higher for relatively small values of f. The initial message may also include f groups, instead of a single group as in DIST-UNI-AWARE.

Another improvement is to have the client send information on its objects when it requests that a group be split. To illustrate, let us consider once again the example of Figure 5.12. At the end of Round 1, the client asks the store to split the initial group into two. Instead, the client could split *its* objects into two groups, TR1 through TR4, and TR5 through TR9, and send signatures for them. In this way, the store could find out that objects TR1 through TR4 exist unchanged in the client, so it then splits the other group into two, thereby saving one round. (That is, the store goes directly to Round 3.)

Another possible optimization is to attempt to "cluster" the changed objects into small groups. For example, suppose that most of the changes at the store are appends of new objects. Then, to group the objects, we first sort them by timestamp (as opposed to by signature, as done by function Group in Figure 5.11). Thus, most changed objects will be contiguous as we break them into groups, implying that with little effort we can pinpoint the few changed groups, and for each send more detailed individual object signatures. This is much more effective that the original scheme, where changed objects are randomly distributed across the sorted objects. It is also important to notice that even though this scheme uses timestamps, its correctness does not depend on correct timestamps. That is, if the timestamps are incorrect (and even different than those the client has for the same object), the algorithm will still work because it detects differences using signatures, not timestamps. Timestamps are only used as "hints" for clustering the objects to reduce the data exchanged. Figure 5.13 illustrates this idea.

In Figure 5.13, the contents of the store and client are shown at the top, with the timestamps indicating when objects were last modified. Initially, the store orders all its objects by timestamp, and forms the initial group with all objects. The descriptor for the group includes the timestamps of the minimum and maximum objects. When the client notices that the signature for this first group does not match what it has, it asks the store to split the group. In Round 2 the store forms two groups and sends the descriptors. Because the objects were sorted by timestamp, all the changes are located in the second group. If that group had to be split, again we would expect changes to be localized. (In the example,the group does not need to be split since the algorithm terminates quickly.) In general, ordering by timestamp tends to avoid

Data Store

Handle	Title	TS
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3
TR4	Paper 4	T4
TR7	Paper 7	T7
TR8	Paper 8	T8
TR9	Paper 9	T9
TR6	Paper 6'	T10

Client

Handle	Title	TS
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3
TR4	Paper 4	T4
TR5	Paper 5	T5
TR6	Paper 6	T6
TR7	Paper 7	T7
TR8	Paper 8	T8

Round 1:

Vectors from Store:

NULL	TR1,T1	TR6,T10	NULL	s(Paper1..Paper9)

Equivalent Vectors in Client:

NULL	TR1,T1	TR6,T10	NULL	s'(Paper1..Paper9)

Client concludes:

LEARN TR6 and start round 2 as s(Paper1..Paper9) $\neq$ s'(Paper1..Paper9).

Round 2:

Vectors from Store:

TR1,T1	TR2,T2	TR4,T4	TR7,T7	s(Paper1..Paper4)
TR4,T4	TR7,T7	TR9,T9	TR6,T10	s(Paper7..Paper9)

Equivalent Vectors in Client:

TR1,T1	TR2,T2	TR4,T4	TR7,T7	s(Paper1..Paper4)
TR4,T4	TR7,T7	TR9,T9	TR6,T10	s(Paper7..Paper9)

Client concludes:

FORGET TR5, LEARN TR9 and no more iterations are needed as both pairs of signatures are equal.

Figure 5.13: Example of the Clustering Technique

having to split many groups, reducing the communication traffic.

This example also illustrates another improvement. Since the store sent as end points of the first group TR1 and TR6, it does not need to use them as end points in subsequent rounds. Thus, in Round 2, the groups are TR2 through TR4, and TR7 through TR9. By sending these smaller groups the algorithm can terminate in Round 2.

5.7 Discussion

In this chapter we have studied awareness mechanisms for repositories of digital objects. We have argued that it is important to cleanly separate the storage functionality from the rest (indexing, replication, and so on), so that many and varied components can use the stores. With such a split, it is critical to have effective awareness schemes that can work correctly even as the objects in a store migrate from one "environment" to another (e.g., as operating systems, file systems and hardware gets upgraded). The signature or content-based schemes we presented are especially well suited for such a scenario, and we believe that the performance enhancements we discussed can make them practical.

We also believe it is instructive to view awareness schemes as variants of unifying algorithms such as UNI-AWARE or DIST-UNI-AWARE. This makes differences in assumptions and performance very apparent. Furthermore, such a unified view of awareness mechanisms could be extremely important for a client that must deal with stores that implement different schemes.

Part III

Efficient Content Access in a Federation of Archival Repositories

Chapter 6

Efficient Searching in an AR Federation

6.1 Overview

A federation of ARs needs not only to preserve information for long periods of time, but it also needs to provide access to the information. In this chapter, the focus is to help users to efficiently find documents with content of interest across a potential set of sources.

There are many mechanisms for searching in a federation of ARs, each with their own advantages and disadvantages. These solutions can be classified into three categories: mechanisms without an index, mechanisms with specialized index nodes (centralized search), and mechanisms with indices at each node (distributed search). For example, Gnutella, a highly decentralized federation of nodes, uses a mechanism in which nodes do not have an index and queries are propagated from node to node until matching documents are found. This search mechanism works by flooding the network (or a subset of it) in the hope of finding a match for a query. Although this approach is simple and robust, it has the disadvantage of the enormous cost of flooding the network every time a query is generated.

Centralized-search systems use specialized nodes that maintain an index of the documents available in the federation of ARs. To find a document, the user queries an index node to identify nodes having documents with the content of interest. These central indices may be built with the cooperation of the nodes (e.g., Napster nodes provide a list of available files at sign-in time) or by crawling the P2P network (as in a web search engine). The advantages of a centralized search mechanism is efficiency as just a single message is needed to resolve a query. However, a centralized system is vulnerable to attack (e.g., index sites can be shut down by a court order or a hacker attack) and it is difficult to keep the indices up-to-date.

A distributed-index mechanism, the option we will study in detail in this chapter, maintains indices at each node. These distributed indices need to be small, so instead of using traditional "destination" indices, we use *Routing Indices* (RIs) that give a "direction" towards the document, rather than its actual location. As an illustration, Figure 6.1 shows four nodes A, B, C, and D, connected by solid lines. The document with content "x" is located at node C, but the RI of node A points to neighbor

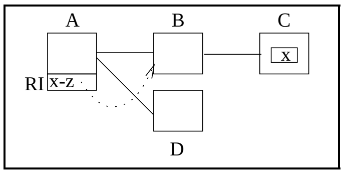

Figure 6.1: Routing Indices

B instead of pointing directly to C (dotted arrow). By using "routes" rather than destinations, the index size is proportional to the number of neighbors, rather than to the number of documents. We can reduce the size of RIs even further by using approximate indices, i.e., by allowing RIs to give a hint (rather than a definite answer) about the location of a document. For example, in the same figure, an entry in the RI of node A may cover documents with contents "x," "y," or "z." A request for documents with content "x" will yield a correct hint, but one for content "y" or "z" will not.

The techniques presented in this chapter can be applied not only to a federation of Archival Repositories but also to more general Peer-to-peer systems (P2P). As in our federation of ARs, in P2P systems, distributed computing nodes of equal roles or capabilities exchange information directly with each other. These systems represent an incredible wealth of information allowing users to exchange documents (Freenet [25]), music files (Napster [94], Gnutella [87]), and even computer cycles (Seti-at-home [91]). Given that a federation of Archival Repositories is a special case of P2P systems, in the rest of the chapter we expand the problem of efficient searching in an AR federation to efficient searching in P2P systems.

In this chapter we study options for building effective RIs, and evaluate their performance. In particular, the contributions of this chapter are:

- We introduce Routing Indices, an efficient way of locating content in a P2P system (Sections 6.3 and 6.4).
- We present three RIs: the compound, the hop-count, and the exponential routing indices (Sections 6.5).

- We evaluate the performance of RIs via simulations, and find that RIs can improve performance by one or two orders of magnitude over a flooding-based system, and by 50-100% versus a random forwarding system (Section 6.7).

6.2 Related Work

The problem of indexing a P2P network is related to the problem of indexing a distributed database [41]. However, algorithms for indexing distributed databases make two fundamental assumptions that are not applicable to P2P systems: that nodes are stable and connected most of the time, and that the number of nodes is small.

There are several working P2P systems currently available, each with its own "indexing" approach. Napster [94] uses centralized indices, which, as stated before, are vulnerable to attack. Gnutella [87] does not build indices; instead, queries flood a significant part of the network, resulting in a simple but very costly approach since just one query can expand into hundreds of thousands of requests through the Gnutella network. Freenet [25] uses an interesting approach to indexing. Each node builds an index with the location of recently requested documents, so if they are requested again, the document can be retrieved at a very low cost.

There are a number of P2P research systems (CAN [65], Oceanstore [43], CHORD [78], Pastry [71], and Tapestry [97]) that can efficiently find documents in a P2P network. The key differences between these systems and our approach is that we do not mandate a specific network structure and that queries are on the content of the documents rather than on document identifiers.

Selecting a neighbor for forwarding a query is also related to traditional routing algorithms [80] such as Bellman-Ford [7, 24]. The major difference with our algorithms is that standard routing algorithms are designed to transmit a packet between two nodes through the shortest route. In our case, we need to get a "packet" from one node to one or *more* nodes so that we can find the best answers to a query. Also, the destination of a packet is not pre-defined (as in IP routing), but instead it depends on the query contained by the packet. IP routing to multiple destinations (multicast algorithms) has been studied extensively (see for example [56]). However, multicast algorithms require the creation of a relatively stable multicast tree.

The problem of selecting the best database to which to send a query was studied as part of the GlOSS project [33, 30]. However, GlOSS assumes that we are selecting among a set of databases, rather than among "paths" that lead to a set of databases.

Some recent work has empirically evaluated P2P systems. A survey and evaluation of centralized-search P2P systems can be found at [95]. An evaluation and description of the present state of Gnutella can be found at [14]. Finally, [96] focuses on search techniques that do not use indices, although it also studies one type of "local area index." In such indices, a node indexes the content of nodes within "r" hops. However, these indices are not routing indices, they are traditional indices.

6.3 Peer-to-peer Systems

A P2P system is formed by a large number of *nodes* that can join or leave the system at any time and that have equal capabilities. Each node is connected to a relatively small set of neighbors which in turn is connected to more nodes. In Figure 6.2, the neighbors of node A are nodes B, C, and D. Note that there might be cycles in the network (such as the one caused by the link between E and G). Each node has a local document database that can be accessed through a local index. The local index receives content queries (e.g., a request for documents containing the words "database systems," a request for documents containing a picture of the sun, etc.) and returns pointers to the documents with the requested content.

6.3.1 Query Processing in a Distributed-Search P2P System

In a distributed-search P2P system, users submit queries to any node along with a stop condition (e.g., the desired number of results). A node receiving a query first evaluates the query against its own database, returns to the user pointers to any results, and, if the stop condition has not been reached, the node selects one or more of its neighbors and forwards the query to them (along with some state information).

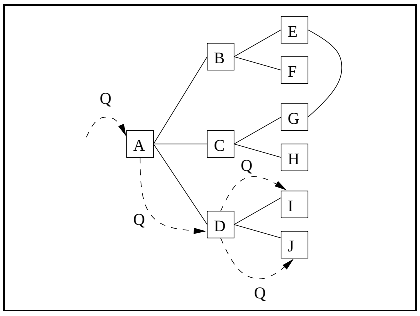

Figure 6.2: P2P Example

In turn, each of the neighbors evaluates the query in a similar fashion, returns result pointers to the user and forwards the query to neighbors.

In Figure 6.2, Node A initially receives a query. Node A checks for local results and sends those results to the requesting node. Then, assuming that the stop condition has not been satisfied, node A selects node D as the best neighbor to handle the query and forwards the query to it (dashed arrow). For nodes to be able to verify if the stop condition has been reached, we need to include the number of results found so far as state information in each query-forwarding message. Then D processes the query and selects I as the best neighbor to continue handling the query. Let us assume now that I has processed the query, but not enough results have been found to reach the stop condition. In this case, I returns the query to D which forwards the query to the next best neighbor (J in this case).

Queries can be forwarded to the best neighbors in parallel or sequentially. A parallel approach yields better response time, but generates higher traffic and may waste resources. In this chapter, we focus on a sequential forwarding of the queries.

6.4 Routing indices

In this section we present an example of how the *compound RI* (CRI) works. Later, in Section 6.5 we present two other RIs: the exponential RI and the hop-count RI.

The objective of a Routing Index (RI) is to allow a node to select the "best"

Path	# docs	Documents with topics: DB	N	T	L
B	100	20	0	10	30
C	1000	0	300	0	50
D	200	100	0	100	150

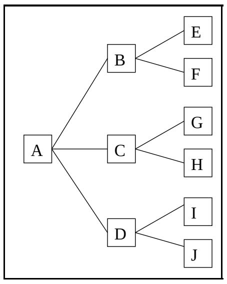

Figure 6.3: A Sample Compound RI

neighbors to send a query to. A RI is a data structure (and associated algorithms) that, given a query, returns a list of neighbors, ranked according to their *goodness* for the query. The notion of goodness may vary but in general it should reflect the number of documents in "nearby" nodes.

As a running example, we will use a P2P system for retrieval of text documents with the network depicted on the right side of Figure 6.3. For simplicity, this network does not have cycles (we discuss cycles in Section 6.6). In this system, documents are on zero or more "topics," and queries request documents on particular topics. Each node has a local index for quickly finding local documents when a query is received. Nodes also have a CRI containing (i) the number of documents along each path and (ii) the number of documents on each topic of interest In Figure 6.3 we show an example of a CRI for node A with three neighbors (paths): B, C, and D. For simplicity, we assume that there are only four topics of interest: databases (DB), networks (N), theory (T), and languages (L). In the figure, we see that we can access 1000 documents through C (i.e., there are 1000 documents in C, G and H) and that of those documents, 300 are about "networks" and 50 are about "languages."

The RI may be "coarser" than the local indices maintained at nodes. For example, node A could maintain a more detailed local index in which each document is further classified into sub-categories. By keeping a *summary* of the detailed index, we achieved a more compact RI at the cost of introducing "errors" when user queries are based on the subcategories. Specifically, the summarizing of the local index may introduce overcounts or undercounts in the RI. For example, a summarization that groups several subtopics into a single topic (e.g., "indices", "recovery", and "SQL" into "databases") may introduce overcounts on the number of documents available. In fact, a query for documents on "SQL" is converted into a query for documents on "databases," making us believe that there are many documents on "SQL" whereas in reality there may be few or even none. Summarization can also introduce undercounts. For example, if the summarization uses a frequency threshold (e.g., throws away topics with very few documents), then we may believe that there are no documents on a topic when there are in fact a few.

Given the index, we need now to compute the "goodness" of each node for a query. For CRIs we will use the number of documents that may be found in a path as a measure of goodness. To compute the number of documents, we will use the estimators in [31, 32]. Given that our focus is not on the estimators but on the use and maintenance of RIs, throughout the chapter we will use a simplified model in which queries are a conjunction of subject topics, documents can have more than one topic, and document topics are independent. Thus, we can estimate the number of results in a path as: $NumberOfDocuments \times \prod_i \frac{CRI(s_i)}{NumberOfDocuments}$ where $CRI(s_i)$ is the value for the cell at the column for topic s_i and at the row for a neighbor.

To illustrate, let us assume that A receives a query for documents on "databases" and "languages." We estimate the number of results as $\frac{20}{100} \times \frac{30}{100} \times 100 = 6$ at B, $\frac{0}{100} \times \frac{0}{100} \times 100 = 0$ at C, and $\frac{100}{200} \times \frac{150}{200} \times 200 = 75$ at D. Therefore, the "goodness" of path B will be 6; of path C, 0; and of path D, 75. These numbers are just estimates and are subject to overcounts and/or undercounts. In particular, if there is a strong correlation between the topics "databases" and "languages," then path B may have as many as 20 documents matching the query for the topics "databases" and "languages" On the other hand, if there is a strong negative correlation between the topics "databases" and "languages," then there may be no documents in path B on either topic.

One limitation of using CRIs is that they do not take into account the difference in cost due to the number of "hops" necessary to reach a document. For example, the documents along path B may all be just one hop away, while the documents along path C may be scattered in a long chain of nodes and finding them would require

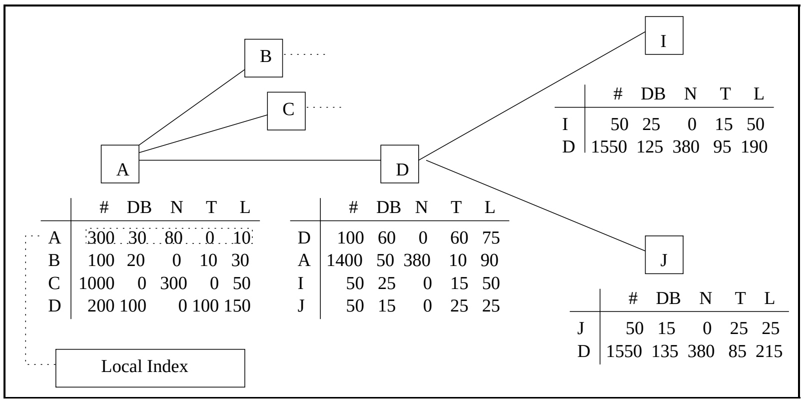

Figure 6.4: Routing Indices

many messages. Later, we will introduce more sophisticated RIs that do not have this limitation.

In the rest of the section we describe how compound RIs are used, created, and maintained.

6.4.1 Using Routing Indices

In this subsection we show how RIs, and in particular compound RIs, can improve the performance of query processing in a P2P system. Consider the P2P system described in Figure 6.3. In Figure 6.4 we present part of the P2P network with RIs attached to each node. For compactness, we represent the four topics of interest: database, network, theory, and languages with the letters DB, N, T, and L respectively. In the example, we assume that the first row of each RI contains a summary of the local index. (This summary can be obtained by consolidating subtopics into the main topics, or perhaps by using clustering on a local keyword index to generate topics for each of its documents.) In particular, the summary of A's local index shows that A has 300 documents: 30 about databases, 80 about networks, none about theory, and 10 about languages. The rest of the rows represent a compound RI. In the example, the RI shows that node A can access 100 database documents through D (60 in D, 25 in I, and 15 in J).

When A receives from a client a query for documents about "databases" and "languages," it first uses the local database to answer the query. If not enough answers are found, it computes the goodness of each path as explained earlier. In this case, the goodness of B, C, and D is 6, 0, and 75 respectively, so A selects D as the best neighbor to forward the query to. In turn, D returns all local results to the client of A and, if not enough results are found, computes the goodness of I and J (25 and 7.5). Since I has the highest goodness, D forwards the query to I. In turn, I returns local results, but it cannot forward the query any further, so (if more results are needed) it returns the query to D which forwards it to its best next neighbor J. Even though the network in the example is very small, a query with a stop condition of 50 documents will generate 9 messages when using flooding, but only 3 messages if using the RI. Even if we send the query serially in a depth-first fashion to neighbors ranked randomly, we will have 3 messages in the best case and 9 messages in the worst case. The savings in the number of messages when using RIs is the result of forwarding the query only to the nodes that have a high potential of obtaining results.

The storage space required by an RI in a node is modest as we are only storing index information for each neighbor. Furthermore, the storage space per neighbor can be adjusted by increasing or decreasing the level of summarization of the index. Specifically, if s is the counter size in bytes, c is the number of categories, N the number of nodes, and b the branching factor (i.e., number of neighbors), then a centralized index would require $c \times (t+1) \times N$ bytes, while each node of a distributed system would need $c \times (t+1) \times b$ bytes. Thus, the total for the entire distributed system is $c \times (t+1) \times b \times N$ bytes. Although the RIs require more storage space overall than a centralized index, the cost of the storage space is shared among the network nodes.

6.4.2 Creating Routing Indices

Let us now turn our attention to how RIs are created. Returning to our running example, let us assume that initially there is no connection between A and D. The initial state of the system is shown by the solid lines of Figure 6.5a. When the $A - D$ connection is established, node A informs node D of all the documents that can be accessed through node A. Specifically, node A *aggregates* its RI and sends it to D. In our example, the aggregation is done by adding all the vectors in the RI. Thus, A sends D a vector saying that it has access to 1400 documents $(300 + 100 + 1000)$, of which 50 are on databases $(30 + 20 + 0)$, 380 on networks $(80 + 0 + 300)$, 10 on theory $(0 + 10 + 0)$, and 90 on languages $(10 + 30 + 50)$. A does not need to send more information as D does not need to know the precise location of the documents, but only that they can be accessed through A. After D receives the aggregated RI from A, it adds an additional row to its RI with A's identifier and A's aggregated RI (as shown in Figure 6.5b). By aggregating RIs, we reduce both the amount of information transmitted and the storage space used. Similarly, D aggregates its RI (excluding the row for A if it is already in the RI), and sends its aggregated RI to A. Additionally, the RI creation process at A and D can be done in parallel.

After A and D update their RIs, they need to inform their other neighbors that they now have access to more documents. Thus, D sends an aggregate of its RI to I (excluding I's row) and to J (excluding J's row) as shown in Figure 6.5b. Then I and J update their RI by replacing the row for D with the new information (not shown in the figure). If I and J were connected to nodes other than D, they would have to send an update to those nodes too.

6.4.3 Maintaining Routing Indices

The process of maintaining RIs is identical to the process used for creating them. To illustrate, let us suppose now that client I introduces two new documents about "languages" in its database. To update the RIs of its neighbors, I summarizes its new local index, aggregates all the rows of its compound RI (excluding the row for D), and sends this information to D. Then D replaces the old row for I with the received aggregated RI. In turn, D computes and sends new aggregates to A and J. When receiving the update, A and J update their RIs and compute new aggregates for their neighbors, and so on. For efficiency, we may delay exporting an update for

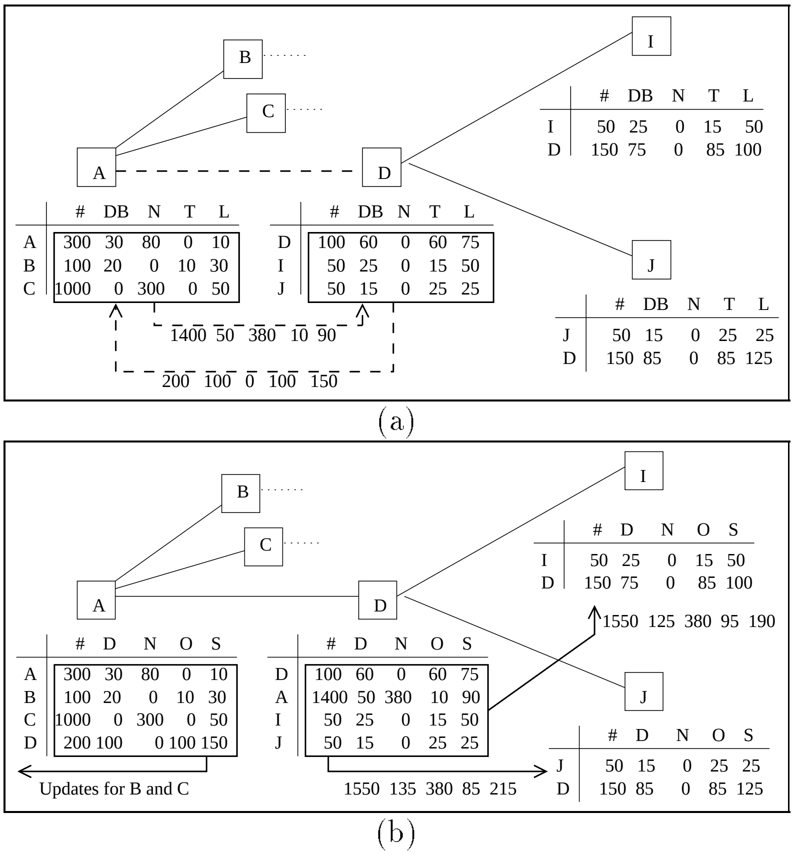

Figure 6.5: Creating a Routing Index

a short time so we can batch several updates, thus trading RI freshness for a reduced update cost. We may also choose not to send updates when the difference between the old and the new value is not significant. By not sending minor updates, we can again trade reduced update cost for accuracy of the RI.

Finally, a special but frequent update case occurs when a node disconnects from the network. To illustrate, let us suppose that I disconnects from the network. Node D detects the disconnection and updates its RI by removing the row for I. Then D informs its neighbors of the change on the number of documents it can access by sending new aggregates of its RI to them. In turn, the neighbors of D update their RIs and propagate the new information to their neighbors. Thus, we did not need I's

	1 Hop					2 Hops				
Node	#	DB	N	T	L	#	DB	N	T	L
X	60	13	2	5	10	20	10	10	4	17
Y	30	0	3	15	12	50	31	0	15	20
Z	5	2	0	3	3	70	10	40	20	50

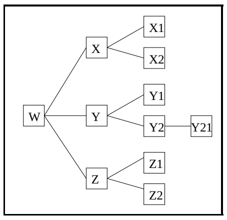

Figure 6.6: A sample Hop-count RI for node W

participation (or the participation of any other neighbor) in the disconnection process. Not requiring the participation of a disconnecting node is an important feature in a P2P system in which nodes can come and go at will.

6.5 Alternative Routing Indices

6.5.1 Hop-count Routing Indices

In this subsection, we present an alternative data structure for an RI: a hop-count RI. The main limitation of the compound RI is that it does not take into account the number of "hops" (query forwardings) required to find documents. In the hop-count RI we stored aggregated RIs for each "hop" up to a maximum number of hops. We call this number the *horizon* of the RI. We show in Figure 6.6 a sample hop-count RI with a horizon of 2 hops. The node with this hop-count RI has three neighbors: X, Y, and Z. With one hop via neighbor X, the node can find 60 documents, out of which 4 are about databases, 2 about networks, 5 about theory, and 10 about systems. The node can also find 20 more documents through X with 2 hops (i.e., at X's neighbors). We do not have information beyond the horizon with this kind of RI.

The estimator of a hop-count RI needs a cost model to compute the goodness of a neighbor. For example, neighbor X may be preferable to neighbor Y for a query on topic "DB," as through X we would find 13 results with one hop, while it would require two hops to find that many results through Y. On the other hand, we can find more results (31) when going through Y.

If we define cost in terms of number of messages (we will expand on the notion of cost in Section 6.7), then we can define the goodness of a neighbor as the ratio between the number of documents available through that neighbor and the number of messages required to get those documents. So a neighbor that allows us to find 3 documents per message is better than a neighbor that allows us to find 1 document per message.

A simple model that allows us to compute this ratio is the regular-tree cost model. The model assumes that document results are uniformly distributed across the network and that the network is a regular tree with fanout F. Under these assumptions, it takes F^h messages to find all documents at hop h. Therefore, we can compute the number of documents per message by dividing the expected number of result documents at each hop by the number of messages needed to find them. Formally, we define the goodness ($goodness_{hc}$) of $Neighbor_i$ with respect to query Q for hop-count RI as: $goodness_{hc}(Neighbor_i, Q) = \sum_{j=0..h} \frac{goodness(N_i[j],Q)}{F^j}$, where h is the horizon of the hop-count RI, $goodness()$ is the estimator for CRI, and $N_i[j]$ is the RI entry for j hops through $Neighbor_i$. In our example, if we assume $F = 3$, the goodness of X for a query about "DB" documents would be $13 + 10/3 = 16.33$ and for Y would be $0 + 31/3 = 10.33$, so we would prefer X over Y.

Let us turn now to the problem of how to create and update hop-count RIs. A node that needs to notify its neighbors of an update in its database first builds the aggregated RI in the same fashion as the compound RI. Then it shifts the columns to the right, so the entries for 1 hop become the entries for 2 hops, the entries for 2 hops become those for 3 hops, and so on. The entries in the last column of the original RI are discarded and the summary of the local index is placed as the first column of the new table (i.e. as the one-hop entry). This new aggregated RI is sent to all neighbors which in turn update their RIs.

6.5.2 Exponentially aggregated RI

The hop-count RI is effective in taking into account the number of hops. However, this benefit comes at a higher storage and transmission cost than the compound

Path	#	DB	N	T	L
X	66.67	16.33	5.33	6.33	15.67
Y	46.67	10.33	3.00	20.00	18.67
Z	28.33	5.33	13.33	9.67	19.67

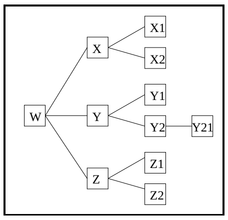

Figure 6.7: A sample Exponential Routing Index for Node W

RI. Moreover, in Section 6.7.3, we will see that the hop-count RI performance is negatively affected by the lack of information beyond the horizon (a hybrid CRI-HRI overcomes this disadvantage, but it still does not solve the storage and transmission cost problem). In this subsection we present an alternative index structure, the exponential aggregated RI, that overcomes these shortcomings at the cost of some *potential* loss in accuracy.

The exponentially aggregated RI stores the result of applying the regular-tree cost formula to a hop-count RI. Specifically, each entry of the ERI for node N contains a value computed as: $\sum_{j=1..th} \frac{goodness(N[j],T)}{F^{j-1}}$, where th is the height and F the fan-out of the assumed regular tree, $goodness()$ is the Compound RI estimator, $N[j]$ is the summary of the local index of neighbor j of N, and T is the topic of interest of the entry.

We show in Figure 6.7 an exponentially aggregated RI computed from our sample network of Figure 6.6. In the figure, we assume that the neighbors of X, Y, and Z are leaf nodes and that the fan-out of the tree is 3. The entries for topic "DB" for X and Y have the values $13 + 10/3 = 16.33$ and $0 + 31/3 = 10.33$.

The exponential RI makes the same assumptions as the regular-tree cost model and may not be realistic in some configurations, but it can still be used as an approximate index. There is a fundamental difference between the exponential RI and the hop-count RI. While the hop-count RI does not have any information beyond the horizon, with the exponential RI we can keep information for all nodes accessible from each neighbor in the RI. In fact, we will see in Section 6.7.3 that the exponential RI outperforms the hop-count RI in most cases.

To update exponential RIs, the node that needs to send an update to a neighbor adds up all rows (except the one associated with the neighbor to which the update vector is sent), multiplies the resulting vector by $1/F$, and adds the goodness of the summary of its local index. Note that multiplying by $1/F$ has the same effect as the shifting of the columns in the hop-count RI, as it does the adding of the goodness of local index with respect to placing the summary of the local index as the first hop. When the neighbor receives this update vector, it replaces the row of the sending node with the new vector and propagates the updates to its neighbors. In updating exponential RIs, it is essential to propagate updates only when the new and old values of the vector are "different" enough (for example by requiring that the Euclidean distance between the two vectors is greater than a certain number). If we do not do this, the exponential RI would propagate updates to all nodes in the network.

6.6 Cycles in the P2P Network

In this section we analyze how cycles affect the process of creating and updating RIs as well as strategies to minimize those effects. To illustrate the effect of cycles, we will use the initial setup of Figure 6.3, but with the network depicted in Figure 6.8 (with the cycle $A-B-E-G-C-A$). Let us assume that node A adds to its database two new "theory" documents and sends a new aggregate of its RI to B. Node B sends the update to its E and F neighbors, prompting E to send an update to G, which sends an update to C, which finally sends an update to A. When A receives this update, it actually mistakingly assumes that additional "theory" documents are available via node C, but those additional documents are its own. Worse than that, the update from C will prompt A to send an update to its neighbors, informing them that they can access two more theory documents, creating an infinite loop. There are three general approaches for dealing with cycles:

No-op solution: No changes are made to the algorithms; this solution only works with the hop-count and the exponential RI schemes. In the case of the hop-count

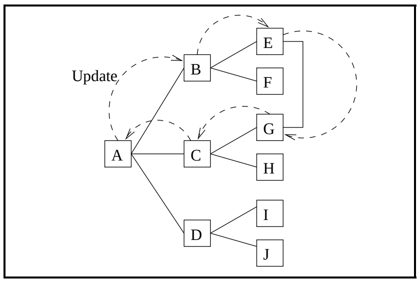

Figure 6.8: Cycles and Routing Indices

RI, cycles longer than the horizon will not affect the RI. However, shorter cycles will affect the hop-count RI but their effect will be limited if we use the regular-tree cost model. However, the cycle increases the cost of creating/updating the hop-count RI as updates sent by a node return to it (via the cycle), causing the node to send a new update to all its neighbors (which in turn send the update back to the node again). The cycle is broken when the update reaches the horizon of the hop-count RI. Similarly, in the case of the exponential RI, updates are sent back to the originator. However, the effect of the cycle will be smaller and smaller every time the update is sent back (due to the exponential decay), until the difference between the old update and the new update is small enough and the algorithm stops propagating the update. As in the hop-count RI, the main effect of the cycle is the increase in cost of creating/updating the RI.

Cycle avoidance solution: In this solution we do not allow nodes to create an "update" connection to other nodes if such connection would create a cycle. The techniques for cycle avoidance have been extensively studied (see [79] for a survey) and we do not cover it further in this chapter. The main disadvantage of this approach is that in the absence of global information we may end up with a suboptimal solution.

Cycle detection and recovery: This solution detects cycles sometime *after* they are formed and, after that, takes recovery actions to eliminate (or neutralize) the effect of the cycles. In the example of Figure 6.8, cycles can be detected by having the originating node of a query or an update, say A, include a unique message identifier in the message. Any update (or query forwarding) that any other node sends as a consequence of this message will have the original message identifier. If a message with the same identifier returns to A (say from C), then A knows that there is a cycle and that a recovery procedure should be started.

6.7 Experimental Results

In this section, we evaluate search mechanisms for P2P systems. First, we present our model of a P2P system. Then we introduce a simulation tool that allows us to evaluate different search mechanisms efficiently. We then use our tool to study the performance of different mechanisms as well as the factors that affect their performance. We close the section with an analysis of the cases where RIs can be used effectively.

6.7.1 Modeling search mechanisms in a P2P system

Our goal in this subsection is to identify the elements of a typical search mechanism in a P2P system, so we can model each element and study its impact on performance. A typical P2P system is a network of nodes T in which each node contains a set of documents. Users send requests consisting of a query q and a $stopCondition$ to a node of the P2P system. The objective of the search mechanism is to answer those requests by obtaining a set of documents of size $stopCondition$ that matches the query q. In addition, search mechanisms allow for updates such as the addition of nodes or new documents.

To process queries and updates we use the mechanisms described in the previous sections (CRIs, HRIs, ERIs). For comparison purposes, we add an additional mechanism: *No-RI*. Instead of using an RI to choose the best neighbor to which to forward a query, this search mechanism simply chooses a random neighbor.

To further model the elements of a search mechanism, we need to define the topology of the network, the location of document results, how costs are measured, and cycle policies.

Parameter Name	Description	Value
Network Configuration Parameters		
$NumNodes$	Number of nodes in the network	60000
T	Topology of the P2P network	tree
F	Branching factor for tree topology	4
EL	Extra links for tree+cycle topology	10
o	Outdegree exponent for power law	-2.2088
Document Distribution Parameters		
QR	Total Number of Query Results	3125
D	Document distribution	80/20
RI Parameters		
$Creation_{size}$	Avg. size of creation/update message	1000 b
$Query_{size}$	Avg. size of query message	250 b
$StopCondition$	Number of documents requested	10
H	Horizon for HRIs	5
A	Decay for ERs	4
c	RI Compression	0%
$minUpdate$	min % diff for update propagation	1%

Figure 6.9: Simulation Parameters

The topology of the network defines the number of nodes and how they are connected. In our model, we consider three kinds of network topologies: a tree, a tree with added cycles, and a power-law graph [44]. The first topology, a tree, is of interest because it does not have cycles (a good base case for our algorithms). For the second topology, we start with a tree and we add extra vertices at random (creating cycles) so we can measure the impact of cycles on the search mechanisms. The third topology, a power-law graph, is considered a good model for P2P systems and allows us to test our algorithms against a "realistic" topology [23].

We model the location of document results using two distributions: uniform and an 80/20 biased distribution. Under the uniform distribution all nodes have the same probability of having each document result (nodes can have more than one document result). The second distribution assigns uniformly 80% of the document results to 20% of the nodes (and the remaining 20% of the documents to the remaining 80% of the nodes).

Modeling the cost of a search mechanism is a complex task. We can model the cost based on the resources used in the P2P system (e.g., network, storage space, or processing power) or based on the user experience (e.g., mean query response time, query throughput, or query turnaround time). In current P2P systems, the critical resource is the network [14] as many of the nodes are connected through links with limited bandwidth (e.g., dial-up connections, DSL, and cable). Therefore, in this chapter we focus on the network and use the number of messages generated by each algorithm as a measure of cost. This is not to say that user-based factors (such as response time) are not important, but by focusing on the network we can also improve those factors.

6.7.2 Simulating a P2P System

In this subsection we describe how our simulator works as well as the choice for the base values of the parameters.

The simulator starts by generating a network topology. Then it distributes results among the nodes, picks at random a node that will initially receive the query or update, and creates the necessary RIs. To build the RIs, we do not need to use the full-fledged algorithms presented, as we know from which node the query will be issued. Instead, we use a version of the algorithm that only updates RI entries for neighbors "downstream" from the node picked as the originator of the query. Cycles are possible when creating RIs. We consider two possible cycle policies: no-op solution and detect and recover (we do not consider the cycle avoidance solution as it has the same effect as the detect and recover policy when processing queries and updates). As the name implies, in the no-op solution we do not do anything special to deal with cycles. For the detect and recover policy, nodes keep track of the queries and updates that they have processed. If a query or update reaches a node for a second time (due to a cycle) the message is not forwarded any further (breaking the cycle).

We generate errors to simulate approximate indices in two ways. In one scenario, we assume that RIs were implemented as hash tables with document counts in each bucket. We generate overcounts by consolidating buckets of the original hash tables and adding their document counts (under the motivation that by consolidating buckets we reduce the size of the index). In the second scenario, errors are generated by using a normal distribution with mean 0 and variance v (making errors positives for overcounts, negative for undercounts, and unchanged for mixed). In all cases, errors are propagated on updates, so they compound from node to node (as they would do in a real system).

In the case of a query, after building the network, distributing the documents, and building the RIs, we send a message to the selected initial node with the number of documents requested ($stopCondition$ in this case). The node finds how many local results are available (if any), subtracts the number of local results from the number of requested documents and if this number has not reached zero, the node sends the request sequentially to each of its neighbors in the order prescribed by the RI (with the number of requested documents set to the number received minus the number of documents found). In turn, a node that receives a forwarded request performs the same procedure until the number of requested documents reaches zero (we are done) or the node does not have more neighbors to which to forward the query (we cannot proceed further). In the latter case, the query is returned to the neighbor that sent the query to that node, and the neighbor sends the returned query to its next best neighbor according to the RI. During this process we count the three kinds of messages that are generated: forwarded queries (when a node sends the query to a neighbor), returned queries (when a node sends back a query because it cannot be forwarded any more and the stop condition has not yet been met), and result messages (sent to the originator of the query informing them that a result can be accessed at a node). We are not counting the actual transmission of the answers as these messages are very application dependent and they do not depend on the performance of the search mechanism. We handle cycles during query processing in the same way as when we built RIs.

In the case of an update, the initially selected node updates its RI, aggregates all rows and sends them to all of its neighbors (appropriately modified depending on the RI used). In turn, each neighbor checks if the changes in the RI are "significant" by comparing the percentage difference between the old and the new value against the *minUpdate* parameter. If the changes are significant, it updates its RI, aggregates all columns and sends the aggregated vector to its neighbors. The stop condition for an update varies among the different RIs as described in previous sections. As before, during the update process, we apply the cycle policies defined in the model.

In the remainder of this section, we explain the choice for base values of the parameters in Figure 6.9. The first parameter, *NumNodes* defines the size of the P2P network. In our simulations, we set $NumNodes$ to 60000, roughly double the maximum number of nodes found in a 2000 study of the Gnutella network [14]. Parameter T defines the topology of the network. To further define the three topologies in our model, we define F as the branching factor for the tree topology, EL as the number of links added to a tree to form a tree with cycles, and o as the outdegree exponent of a power-law network. In the simulator, we generate a power-law network by using the power-law out-degree generator [61]. We set the outdegree exponent to -2.20288 equal to the best fit for the Internet [23]. The Document Distribution Parameters define how documents are distributed throughout the P2P system. For simplicity, we assume that all queries have the same number of results (QR). Reference [95] found that about 5.2% of the nodes of the Gnutella network will have an answer for a given query, so we set this number to 3125 (5.2% of 60000 nodes). Parameter D defines the location of document results in the network. Finally, the RI Parameters define the size of RI tables and messages as well as the parameters needed for each RI algorithm.

6.7.3 Evaluating P2P Search Mechanism

In this section we experimentally compare the three proposed RIs: compound RI (CRI), hop-count RI (HRI), and exponential RI (ERI) against each other and against the No-RI search mechanism. We also explore how the performance of the RIs are affected by approximate indices, different stop conditions, document result distribution, number of document results, and network topology. All results were computed with at least a 95% confidence interval of having a relative error of 10% or less. Parameters are set to the base values presented in Figure 6.9 unless stated otherwise.

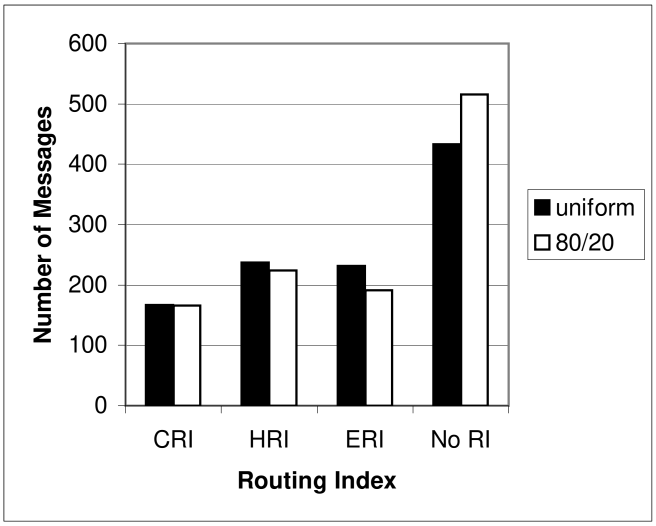

Figure 6.10: Comparison of CRI, HRI, and ERI

Figure 6.10 shows the number of messages needed to process a query when using each kind of RI for two document distributions. The advantage of using RIs is obvious; we are able to reduce the number of messages by half when compared to not using RI. Among the RIs, CRI had the best performance, followed by the ERI and HRI. This difference in performance is a function of the number of nodes used to generate the index. In particular, CRI uses all nodes in the network, HRI uses nodes within a predefined horizon, and ERI uses nodes until the exponentially decayed value of an index entry reaches a minimum value (resulting in using more nodes than HRI, but fewer than CRI). This result shows that the more nodes an RI uses to compute the goodness of a path, the better the RI is. However, we will see that a larger number of nodes implies a higher update cost.

In Figure 6.10 we also present the effect of using two document distributions, an 80/20 biased and a uniform document distribution. Surprisingly, a 80/20 biased distribution does not improve the performance of RIs much, but it degrades the performance of a No-RI search mechanism. To understand this result, we analyzed traces of our simulations. In the case of RIs under a 80/20 document distribution, the algorithm directed the queries to nodes with a high number of document results, but to reach those content-loaded nodes the queries needed to travel through several nodes that had very few or no document results. On the other hand, under a uniform document placement, the algorithms followed good paths where at each node it obtained a few results. In summary, in one case, we collected results by traveling over an almost empty path to a full node, while in the other case, we collected results by traveling through a path of a similar length in which each node contributes a few documents. The overall result was that the number of messages per document result was about the same in both cases. In the case of the No-RI approach, an 80/20 document distribution penalizes performance as the search mechanism needs to visit a number of nodes until it finds a content-loaded node (generating a large number of messages in the process).

We also compared RIs against non-index/flooding solutions such as Gnutella. In that case, RIs reduce the number of messages by up to two orders of magnitude (see Figure 6.12). However, this comparison is not completely fair as non-index systems find all results (versus only a user-defined number of results when using RIs) and they potentially have a better response time (as queries are processed in parallel, rather than sequentially). However, finding *all* results may be an overkill for most applications as users rarely examine more than the first 10 top results returned by a search engine [11]. In addition, low response times may be hard to achieve in Gnutella-like systems because of network contention created by tens of thousands of messages generated by each query.

We studied how increases in the requested number of documents affects RIs. In Figure 6.11, we observed, as expected, that the higher the number of requested documents, the more messages generated. However, the increase in the number of messages is linear for all RIs, demonstrating that they scale well on this parameter. We also analyzed the effect of a decrease in the number of document results available. Figure 6.12 shows the number of messages needed by the search mechanism when using each kind of RIs and flooding at different levels of "potential" query results (i.e., number of documents with the requested content of interest available in the system).

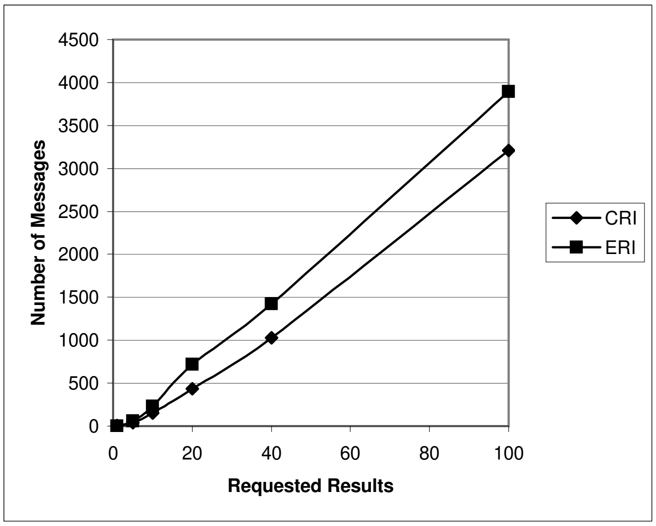

Figure 6.11: Effect of Number of Documents Requested

As expected, the higher the number of documents with content of interest, the better RIs perform. However, as before, the number of messages grows linearly with a reduction of document results (x-axis in graph has a logarithmic scale). The reason for the slight increase in the flooding curve is that the flooding mechanism always find all documents with content of interest (while the search mechanisms based on RI only find 10 documents).

We now investigate how errors in RIs, and particularly overcounts, affect RI performance. As discussed in Section 6.4, errors can occur in a variety of ways; here we select one scenario to illustrate. We assume that documents are organized into categories, and the index is a hash table of categories. Several categories may hash to the same bucket, so the count in a bucket represents the aggregate number of documents in those categories. For example, suppose there are 3 "database" documents, and 2 "network" ones. If "database" and "network" hash to the same bucket, the consolidated bucket will have a count of 5 documents. If a query is looking for "database" documents, when using the RI we will believe that there are 5 of them, when in reality there are only 3 (an overcount). (Instead of adding the original document counts, we

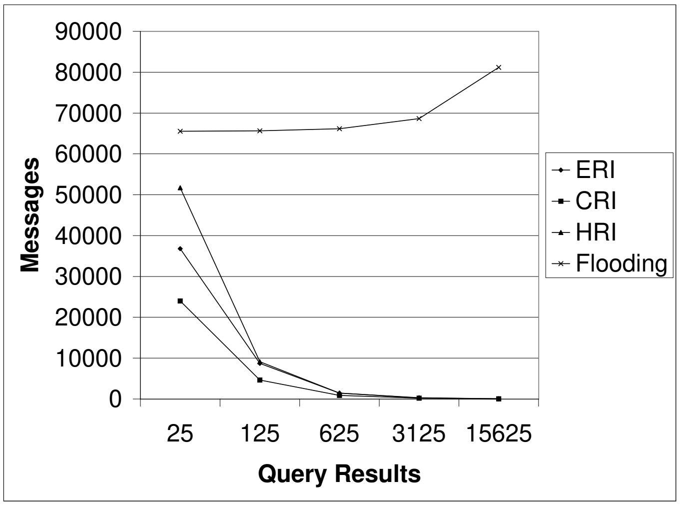

Figure 6.12: Effect of Number of Potential Results

could have also chosen to take the minimum of them, generating undercounts; or we could have averaged them, generating mixed errors.)

As the table size is reduced, more and more overcounts occur. In Figure 6.13, we show the performance of CRI, HRI and ERI, as a function of the "index compression." For example, a 50% value means that the number of hash table buckets is half the number of categories, while 83% represents a table with one-sixth the categories. (Note that the scale is not linear.) From the graph, we can see that even though there is a loss of performance because of overcounts, this loss is modest even in the case of significant reductions in the size of the index. Moreover, query processing when using RIs is still far cheaper than the No-RI approach even if we use the highest compression levels. We conducted additional experiments for undercounts and mixed errors as well as for other error models with results similar to the one presented here.

In Figure 6.14 we study how ERIs perform when cycles are added to a tree network. Cycles are created by adding random links to a tree network with $NumNodes - 1$ links. As expected, the number of messages increases as we add more links and cycles are created. The increase in traffic is the result of two factors. First, there is a loss of accuracy of the RI. In the case of the "detect and recover" policy, this loss is the

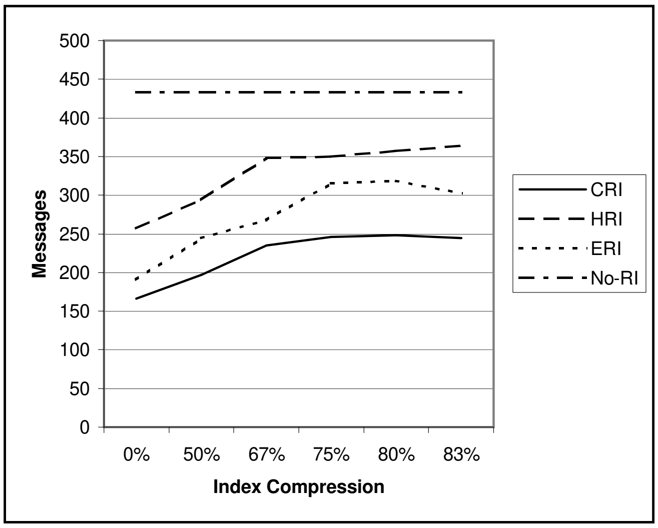

Figure 6.13: Effect of Overcounts

result of missing the best route to the results (as explained in Section 6.6), and in the case of the "no-op" policy accuracy suffers because of overcounts introduced in the generation of the RI. Second, during query processing the number of messages increases. In the case of the "detect and recover" policy, those extra messages are the result of return-queries messages sent by a node that detects a cycle. In the case of the no-op policy, extra messages are generated when we traverse a cycle more than once, finding document results that were already found in a previous iteration. In figure 6.14, we observe that the increase in the number of messages is small if we use the "detect and recover" policy, but it can be significant if we choose to ignore cycles. An unexpected result is that the number of messages drops if we add a large number of links. This drop is the result of the added connectivity that additional links create, which allows shorter routes to document results. Similar performance to the one presented for ERI is shown by HRI and CRI (when using the ignore-detect policy, as CRI is not guaranteed to terminate when using the no-op policy).

In Figure 6.15 we study how RIs perform in different network topologies. The result of our analysis is surprising at first glance: RIs perform better in a power-law

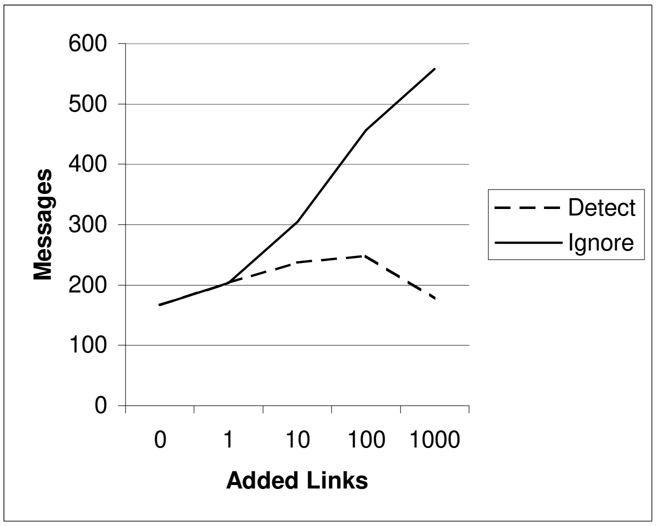

Figure 6.14: Effect of Cycles

network than in a tree network. There are two reasons for this result. First, while in a tree-like network the connectivity of every node (except leaf nodes) is the same, in a power-law network a few nodes have a significantly higher connectivity than the rest. By analyzing the traces of our simulation, we found that the query algorithms actually direct the queries towards those well-connected nodes. After getting to these highly connected nodes, a large number of results is collected without having to issue many messages. The second reason for this performance improvement is that power-law distributions generate network topologies where the average path length between two nodes is lower than in tree topologies. Lower path length improves performance as we need fewer messages to go from node to node. On the other hand, these same two factors hinder the performance of the No-RI approach. In a power-law network, there are very few highly connected nodes and it is not easy to find them if we just move randomly as No-RI does. As a result, the No-RI approach visits a significant number of nodes until it finally stumbles onto a highly connected node (generating a large number of messages in the process). Shorter path length also hinders No-RI as bad decisions about which neighbor to contact often result in return-query messages.

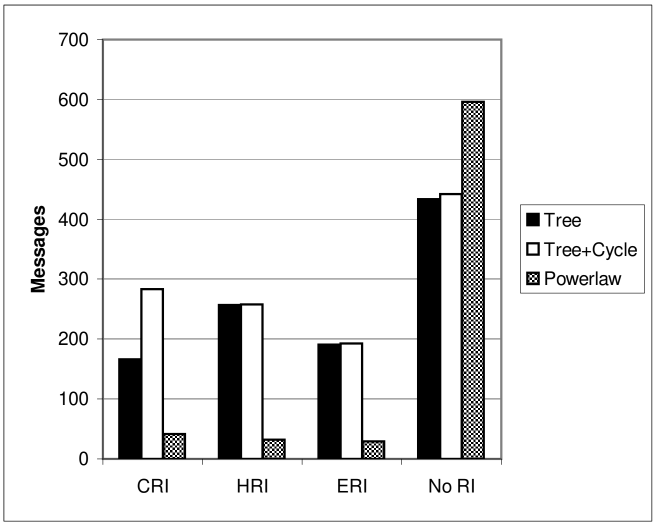

Figure 6.15: Network topology

To further illustrate why shorter path length helps our algorithm, let us consider two extreme cases: a linear network, in which each node N_i is connected only to nodes N_{i+1} and N_{i-1}, and a star network in which all nodes connect to a single node (note that the number of links is the same as in the linear network). In the first case, the RI is not helping much as each node only has two neighbors and to find all documents we would have to jump from node to node, generating a large number of messages. In the second case, however, we can find results very quickly as the RI of the node in the center of the star will forward queries directly to the nodes with document results.

Figure 6.16 shows the number of messages needed to update each kind of RI for each network topology. The graph shows the cost of one batch of updates, propagated throughout the network. In the graph we can see that the cost of CRI is much higher when compared with HRI and ERI. This is the result of CRI propagating the update to all nodes, while HRI and ERI only propagate the update to a subset of the network. This result confirms that the additional information and better query performance of CRIs come with a high price tag. On the other hand, HRIs and ERIs have very low update costs and their query processing performance is very close to the one of

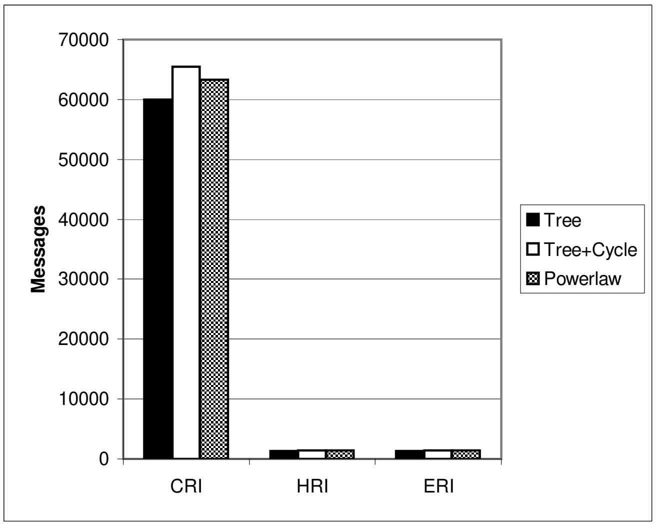

Figure 6.16: Updates and Network Topology

CRIs, making them an excellent choice as the search mechanism of a P2P system. In the graph we can also see that network topology has little impact on the update performance of RIs, as there is low or no correlation between the network topology and the number of nodes that need to be updated.

We also studied the tradeoff between query and update costs for RIs (Figure 6.17). For a system processing 1032 queries per minute (the average query load observed on a section of the Gnutella network [95]), the point where the total cost of using ERIs is the same as the cost of a system without RIs was at an update load of 36 updates per minute. In practice we would expect the number of updates to be way below 36 per minute, especially since it is not that critical to keep indices up to date and updates can be batched together. Thus, the search improvements afforded by RIs are seldom outweighed by the cost of updating them.

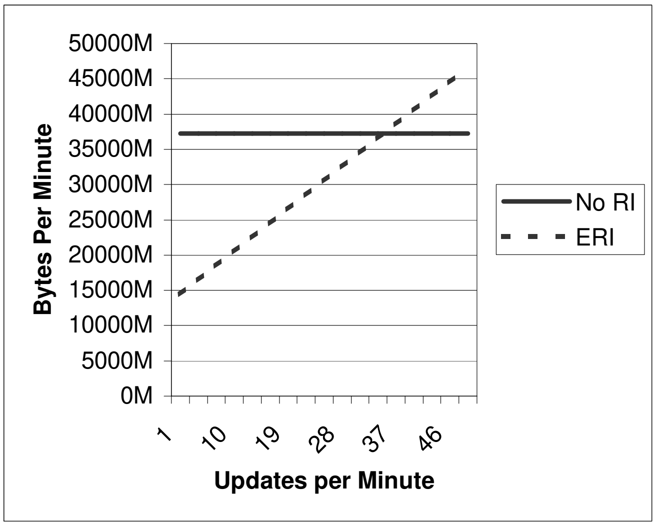

Figure 6.17: Updates vs. Queries

6.8 Discussion

In this chapter we studied how to improve the efficiency of content search in a peer-to-peer system, and in particular, in a federation of ARs. We achieve greater efficiency by placing Routing Indices in each node. Three possible RIs: compound RIs, hop-count RIs, and exponential RIs, were proposed and experimentally evaluated using simulations. From our experiments we conclude that ERIs and HRI offer significant improvements versus not using an RI, while keeping update costs low. We believe that routing indices, and in particular ERIs and HRIs, can help improve the search performance not only of ARs but also of current and future P2P systems.

Chapter 7

Efficient Networks in an AR Federation

7.1 Overview

As we explained in Chapter 6, a federation of ARs needs not only to preserve information for long periods of time, but also needs to provide efficient access to the archived information. In this chapter, we study how to build efficient networks to improve search performance (we will see that efficient networks help most searching mechanism, including the Routing Indices that we presented in Chapter 6). As in the previous chapter, the techniques we propose here can be applied not only to a federation of Archival Repositories but also to more general Peer-to-peer systems (P2P). Therefore, we will tackle the more general problem of searching in P2P systems, while making references to the differences with federations of Archival Repositories when needed.

Search techniques used in current large federated systems are very inefficient and they do not scale well as the number of nodes increases. This inefficiency arises because most of those systems create a random overlay network where queries are blindly forwarded from node to node. As an alternative, there have been proposals for "rigid" systems that place content at nodes based on hash functions, thus making it easier to locate content later on (e.g., [78, 65]). Although such schemes provide good performance for point queries (where the search key is known exactly), they are not as effective for approximate, range, or text queries. Furthermore, in general, nodes may not be willing to accept arbitrary content nor arbitrary connections from others.

In this chapter we propose Semantic Overlay Networks (SONs), a flexible network organization that improves query performance while maintaining a high degree of node autonomy. With Semantic Overlay Networks (SONs), nodes with semantically similar content are "clustered" together. To illustrate, consider Figure 7.1 which shows eight nodes, A to H, connected by the solid lines. When using SONs, nodes connect to other nodes that have semantically similar content. For example, nodes A, B, and C all have "Rock" songs, so they establish connections among them. Similarly, nodes C, E, and F have "Rap" songs, so they cluster close to each other. Note that we do not mandate how connections are done inside a SON. For instance, in the Rap

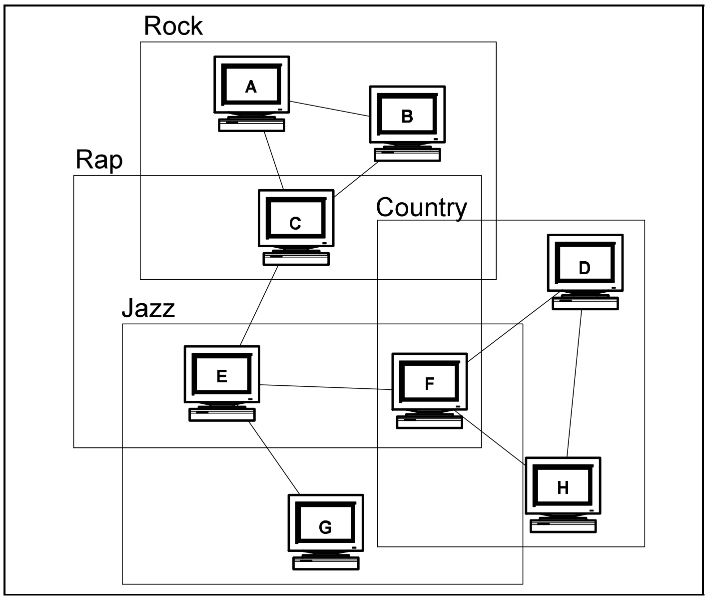

Figure 7.1: Semantic Overlay Networks

SON node C is not required to connect directly to F. Furthermore, nodes can belong to more than one SON (e.g., C belongs to the Rap and Rock SONs). In addition to the simple partitioning illustrated by Figure 7.1, in this chapter we will also explore the use of content hierarchies, where for example, the Rock SON is subdivided into "Soft Rock" and "Hard Rock."

In a SON system, queries are processed by identifying which SON (or SONs) are better suited to answer it. Then the query is sent to a node in those SONs and the query is forwarded only to the other members of that SON. In this way, a query for Rock songs will go directly to the nodes that have Rock content (which are likely to have answers for it), reducing the time that it takes to answer the query. Almost as important, nodes outside the Rock SON (and therefore unlikely to have answers) are not bothered with that query, freeing resources that can be used to improve the performance of other queries.

There are many challenges when building SONs. First, we need to be able to classify queries and nodes (what does "contain rock songs" means?). We need to decide the level of granularity for the classification (e.g., just rock songs versus soft, pop, and metal rock) as too little granularity would not generate enough locality, while too much would increase maintenance costs. We need to decide when a node should join a SON (if a node has just a couple of documents on "rock," do we need to place it in the same SON as a node that has hundreds of "rock" documents?). Finally, we need to choose which SONs to use when answering a query.

Many of our questions can only be answered empirically by studying real content and how well it can be organized into SONs. For our empirical evaluation we have chosen music-sharing systems. These systems are of interest not only because they are the biggest P2P application ever deployed, but also because music semantics are rich enough to allow different classification hierarchies. In addition there is a significant amount of data available that allows us to perform realistic evaluations.

In this chapter we study options for building effective SONs and evaluate their performance by using an actual snapshot of a set of music-sharing clients. The main contributions of this chapter are:

- We introduce the concept of SONs, a network organization that can efficiently process queries while preserving a high degree of node autonomy.
- We analyze the elements necessary for the building and usage of SONs.
- We evaluate the performance of SONs with real user data and find that SONs can find results with only 10%-20% of message overhead that a system based on a random topology would incur.
- We introduce Layered SONs, an implementation of SONs that further improves query performance at the expense of a marginal reduction of the maximum achievable recall level.

7.2 Related Work

The idea of placing data in nodes close to where relevant queries originate was used in early distributed database systems [41]. However, the algorithms used for distributed databases are based on two fundamental assumptions that are not applicable to P2P systems: that there are a small number of stables nodes, and that the designer has total control over the data.

There are a number of P2P research systems (CAN [65], CHORD [78], Oceanstore [43], Pastry [71], and Tapestry [97]) that are designed so documents can be found with a very small number of messages. However, all these techniques either mandate a specific network structure or assume total control over the location of the data. Although these techniques may be appropriate in some application, the lack of node autonomy has prevented their use in wide-scale P2P systems. In addition, the ability to create groups of peers in a P2P system is supported by the JXTA framework [89]. However JXTA (as a framework) does not prescribe the structure of the groups, or when nodes should join a group, or how to search between the groups.

Semantic Overlay Networks are also related to the concept of online communities [76] such as Yahoo Groups [88] and MSN Communities [86]. In an online community, users with common interest join specific groups and share information and files. However, most online communities contain a central element that coordinates the actions of the members of the group.

The problem of how to disseminate queries within a P2P system has been studied recently. The problem of which neighbors a query should be forwarded can be solved by using local indices [96], routing indices(see Chapter 6), or by simply sending the query to all neighbors (as Gnutella does).

There is a large corpus of work on document clustering using hierarchical systems (see [53] for a survey). However, most clustering algorithms assume that documents are part of a controlled collection located at a central database. Clustering algorithms for decentralized environments have also been studied in the context of the world wide web. However, these techniques depend on crawling the data into a centralized site and then using clustering techniques to either make web search results more accurate (as in SONIA [72]) or easier to understand (as in Vivisimo [92]). A more decentralized approach has been taken by Edutella [57] where peers with similar content connect to the same super peer.

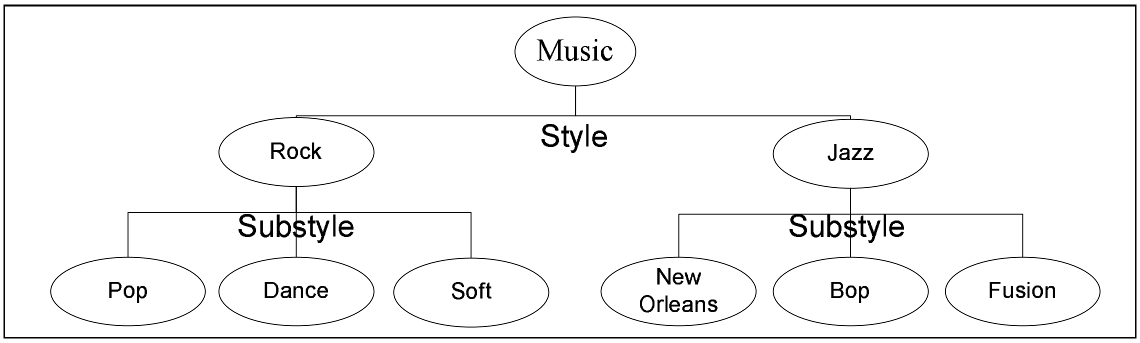

Figure 7.2: A Classification Hierarchy

7.3 Semantic Overlay Networks

In this section we formally introduce the concept of Semantic Overlay Networks (SONs). We model our system as a set of nodes N where each node $n_i \in N$ maintains a set of documents D_i (a particular document may be stored in more than one node). We denote the set of all documents in all nodes as $\mathcal{D}$. Each node is *logically* linked to a relatively small set of nodes (called its neighbors) which in turn are linked to more nodes. A link is a triple (n_i, n_j, l) where n_i and n_j are the connected nodes and l is a string. We call the set of links with the same l, an overlay network. As links are bidirectional, (n_i, n_j, l) and (n_j, n_i, l) are the same.

Current P2P system are established by a single overlay network (i.e., all links have the same l). However, this needs not to be the case and a P2P system can have multiple overlay networks. In this case, a node can be connected to a set of neighbors through an l_1 link and to a potentially different set of nodes through an l_2 link. We will see that a carefully chosen set of overlay networks can improve search performance.

In this chapter, we are focusing on the usage and creation of overlay networks, and not on how queries are routed within an overlay network (see Section 7.2 for a brief overview of current solutions to the intra-overlay network routing problem). Therefore, we will ignore the actual link structure within an overlay network and we will represent an overlay network just by the set of nodes in it ($ON_l = \{n_i \in$

$N | \exists$ a link $(n_i, n_j, l)\})$. In addition, we assume that an overlay network ON_l supports three functions: $Join(n_i, l)$, where one or more links of the form (n_i, n_j, l) are created $(n_j \in ON_l)$; $Search(r, l)$ that returns a set of nodes in ON_l with matches for request r; and $Leave(n_i, l)$ where we drop all the links in ON_l involving n_i.

The implementation of the functions $Join(n_i, l)$, $Search(r, l)$ and $Leave(n_i, l)$ will vary from system to system. Additionally, these functions may be implemented by each node of the network, a subset of it, or even be provided by a computer outside the network. For example, in the Gnutella file sharing system, $Join(n_i, l)$ starts by linking the node n_i to a set of well known nodes (whose addresses are usually published on a web page). Then, n_i can learn about additional nodes (and potentially link to them) by sending "ping" messages through the network (nodes may reply to a "ping" message with a "pong" message that contains their identity). The function $Search(r, l)$ in Gnutella works by having a node send the request along with a "horizon" counter (TTL) to all its neighbors. The neighbors check for matches, returning their identifier to the original requesting node if there are any matches. Then these nodes decrement the horizon counter by one and send the request and the new counter to their neighbors. The process continues until the counter reaches zero, when the request is discarded. Finally, $Leave(n_i, l)$ is implemented in Gnutella by simply dropping all the l links of n_i.

Requests for documents are made by issuing a query q and some additional system-dependent information (such as the horizon of the query). A query is also system dependent and it can be as simple as a document identifier, or keywords, or even a complex SQL query. We model a match between a document d and a query q as a function $M(q, d)$ that returns 1 if there is a match or 0 otherwise. The number of *hits* for a query q in a node n_i is the number of matches in the node $(H(q, n_i) = \sum_{d \in D_i} M(q, d))$. Similarly, the number of hits in an overlay network will be $H(q, ON_l) = \sum_{n_i \in ON_l} H(q, n_i)$. We will denote the probability of a match between q_i and d_j (i.e., $Prob(M(q_i, d_j) = 1))$ as $P_M(q_i, d_j)$. Queries can either be exhaustive or partial. In the first case, the system must return all documents that match the query. In the second case, the request includes a minimum number of results that need to be returned.

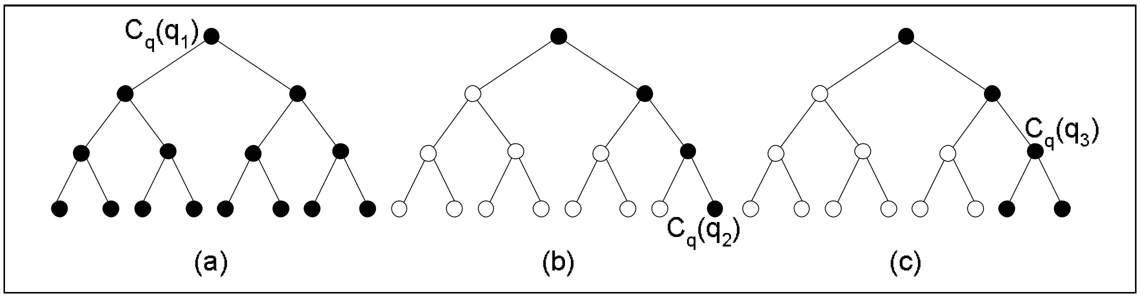

Figure 7.3: Classification Examples

7.3.1 Classification Hierarchies

Our objective is to define a set of overlay networks in such a way that, when given a request, we can select a small number of overlay networks whose nodes have a "high" number of hits (or all hits if the query is exhaustive). The benefit of this strategy is two fold. First, the nodes to which the request is sent will have many matches, so the request is answered faster; and second, but not less important, the nodes that have few results for this query will not receive it, avoiding wasting resources on that request (and allowing other requests to be processed faster).

We propose using a classification hierarchy as the basis of the formation of the overlay networks. A classification hierarchy, H is a tree of concepts such as the one in Figure 7.2. In the figure, we show a classification hierarchy where music documents are classified according to their style (rock, jazz, etc.) and their substyle (soft, dance, etc.).

Each document and query is classified into one or more leaf concepts in the hierarchy. Conceptually, the classification of queries and documents is done by two functions, $C_q^*(q)$ and $C_d^*(d)$ respectively, which return one or more leaf concepts in the hierarchy. These functions are chosen so that if $M(q,d) = 1$ then $C_q^*(q) \cap C_d^*(d) \neq \emptyset$. However, in practice, classification procedures may be *imprecise* as they may not be able to determine exactly to which concept a query or document belongs. In this case, imprecise classification functions, $C_q(q)$ and $C_d(d)$, may return non-leaf concepts, meaning that the document or query belongs to one or more descendants of the non-leaf concept, but the classifier cannot determine which one. For example, when using the classification hierarchy of Figure 7.2, a "Pop" document may be classified as "Rock" if the classifier cannot determine to which substyle ("Pop," "Dance," or "Soft") the document actually belongs. Specifically, if $c \in C_q^*(q)$ then $\exists c' \in C_q(q)$ such that $c' \geq c$, and if $c \in C_d^*(d)$ then $\exists c' \in C_d(d)$ such that $c' \geq c$, where $c' \geq c$ means that c' is equal to c, or that c' is a ancestor of c in H. This definition and the definition of C^* imply that if $M(q,d) = 1$ then $\exists c_q \in C_q(q)$ and $c_d \in C_d(d)$ such that $c_q \geq c_d$ or $c_d \geq c_q$.

Classifiers may also make *mistakes* by returning the wrong concept for a query or document. Specifically, a classifier mistake happens when $M(q,d) = 1$ but $\nexists (c_q \in C_q(q)$ and $c_d \in C_d(d))$ such that $c_q \geq c_d$ or $c_d \geq c_q$. In the following discussion, we will assume that $C_q(q)$ and $C_d(d)$ are imprecise but that they do not make mistakes. However, in our experiments we will study how much the system is affected in the presence of classifier mistakes.

In most systems, document classifications change infrequently, so it is advantageous to classify documents in advance. Then, to speed up searches, documents can be placed in "buckets" that are associated with each concept in the hierarchy. There are two basic strategies for deciding in which bucket a document should be placed: *differential* and *total* assignment. When using a differential assignment, a document d is placed in the bucket of concept c if $c \in C_d(d)$. On the other hand, when using a total assignment, a document is placed in the bucket of concept c if either $c \in C_d(d)$, or c is an ancestor of some element of $C_d(d)$ in H, or c is a descendant of some element of $C_d(d)$ in H. To illustrate, when using the hierarchy of Figure 7.2, a differential assignment of a document classified as "rock" will place it in the bucket associated with the concept "rock", while a total assignment of the same document will place it in the buckets associated with the concepts "rock," "music," "pop," "dance," and "soft."

Given a query q, we now need to decide which bucket (or buckets) need to be considered for finding matches. If we used a total assignment, we consider the buckets associated with each element of $C_q(q)$. On the other hand, if we used a differential assignment, we need to consider a larger set of buckets: the bucket associated with each element of $C_q(q)$, the bucket associated with the ancestors of the elements of

$C_q(q)$, and the bucket associated with the descendants of the elements of $C_q(q)$. As mentioned before, queries can be exhaustive or partial. In the case of an exhaustive query, we need to find all matches, so all buckets that may contain results need to be considered; while in the case of partial queries we do not need to consider all of them. Given that we need to choose among a set of buckets, partial queries add an additional dimension to the problem as now we need to select the best subset of buckets to answer the query. As the exhaustive case is not common in current P2P systems and it is a special case of partial queries, in the rest of the chapter, we will assume that we are answering partial queries. We also assume that document assignment is done using the differential strategy.

Let us now illustrate how classification functions help reduce the number of documents that need to be considered when answering a query. For simplicity, we will assume that the classification functions return a single element of the hierarchy. In Figure 7.3 we present several combinations of classification of documents and queries. In Figure 7.3a, we show the worst-case scenario for our system when a query is classified at the root concept of the hierarchy. This classification indicates that the query results can actually be in any of the leaf concepts in the hierarchy and therefore documents classified in any category in the system can match the query (depicted as black circles in the classification hierarchy). In Figure 7.3b, the query is classified at one of the leaf concepts. In this case, we know that only documents that belong (or may belong) to this concept can match the query; thus, we need to consider the documents classified in that base concept and all the ancestor concepts of it and we can safely ignore all the documents classified into concepts depicted as white circles. Finally, in Figure 7.3c, the query is classified at an intermediate concept in the hierarchy tree. In this case, documents matching the query may belong to any of the descendant leaf concepts, so we need to consider all the descendant concepts of the $C_q(q_3)$, as well as the ancestors of it. In conclusion, given $C_q(q)$, we only need to consider documents for which their $C_d(d)$ is an ancestor of $C_q(q)$ or a descendant of $C_q(q)$. Note that the more precise the classification function $C_q(q)$ is, the smaller the number of concepts that needs to be considered for a match. In addition, the more precise $C_d(d)$ is, the smaller the number of documents that will be classified in the intermediate nodes of the hierarchy, thus also reducing the number of documents that need to be considered for a match.

So far we have considered documents by themselves, but in a P2P system, documents are actually kept by nodes. Therefore, we need to place nodes, rather than documents in buckets. We call a bucket of semantically related nodes a Semantic Overlay Network. Formally, we define a *Semantic Overlay Network* as an overlay network that it is associated with a concept of a classification hierarchy. For short, we will call a SON associated with concept c, simply the SON of c or SON_c. For example, in the hierarchy in Figure 7.2, we will define 9 SONs: 6 associated with the leaf nodes (soft, dance, pop, New Orleans, etc.), one associated with rock, another associate with jazz, and a final one associate with music. To completely define a SON, we need to explain how nodes are assigned to SONs and how we decide which SONs to use to answer a query.

A node decides which SONs to join based on the classification of its documents. Thus, since we are using a differential assignment of documents, a node n_i joins SON_c if there is a $d \in D_i$ such as $c \in C_d(d)$. Under this definition, a query q associated with the concepts $C_q(q)$ will only find results in SON_c where $c \in C_q(q)$ or $c \leq c' \in C_q(q)$ or $c \geq c' \in C_q(q)$. Note that this strategy is very conservative as it will place a node in SON_c if just one document classifies as c. A less conservative strategy will place a node in SON_c if a "significant" number of document classifies as c. Note that a less conservative strategy has two effects: it reduces the number of nodes in a SON and it reduces the number of SONs to which a node belongs. The first of these effects increases the advantages of SONs as less nodes need to be queried. The second effect reduces the cost of SONs as the greater the number of SONs to which a node belongs, the greater the node overhead for handling many different connections. However, a less conservative strategy may prevent us from finding all documents that match a query. In Section 7.6, we study different strategies for assignment of nodes to SONs.

After assigning nodes to SON, we may make adjustments to the SONs based on the actual data distributions in the nodes. For example, if we observe that a SON contains only a very small number of nodes, we may want to consolidate that SON with a sibling or its parent in order to reduce overhead.

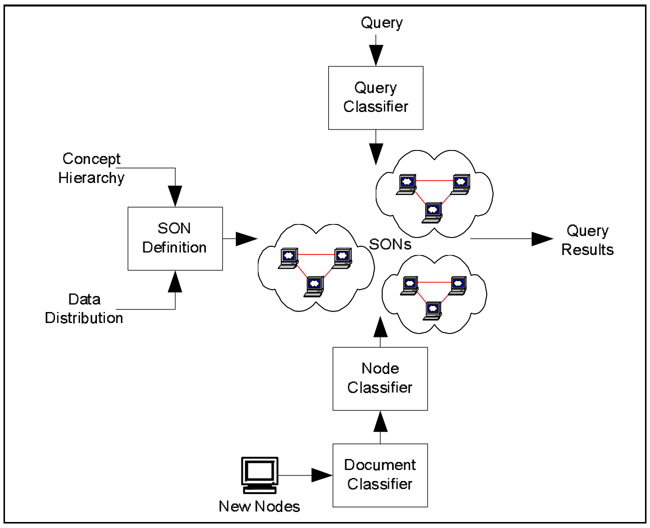

Figure 7.4: Generating Semantic Overlay Networks

To summarize, the process of building and using SONs is depicted in Figure 7.4. First, we evaluate potential classification hierarchies using the actual data distributions in the nodes (or a sample of them) and find a good hierarchy. We use that hierarchy to define the SONs of our system. A node joining the system, first runs a document classifier on all its documents. Then, a node classifier assigns the node to specific SONs (by, for example, using the conservative strategy described in this section). Similarly, when a query is issued, it needs to be classified and sent to the appropriate SONs.

In the next sections, we will study the challenges and present solutions for building a P2P system using Semantic Overlay networks. We will evaluate our solutions by simulating a music-sharing system based on real data from Napster [94] and OpenNap [90]. Specifically, in this chapter we will address the following challenges:

- Classification hierarchies for SONs (Section 7.4): If nodes have very diverse files, there will not be enough clustering to merit the use of SONs. So, in practice, will we see enough clustering? What hierarchies will yield the most clustering and the best SON organization?

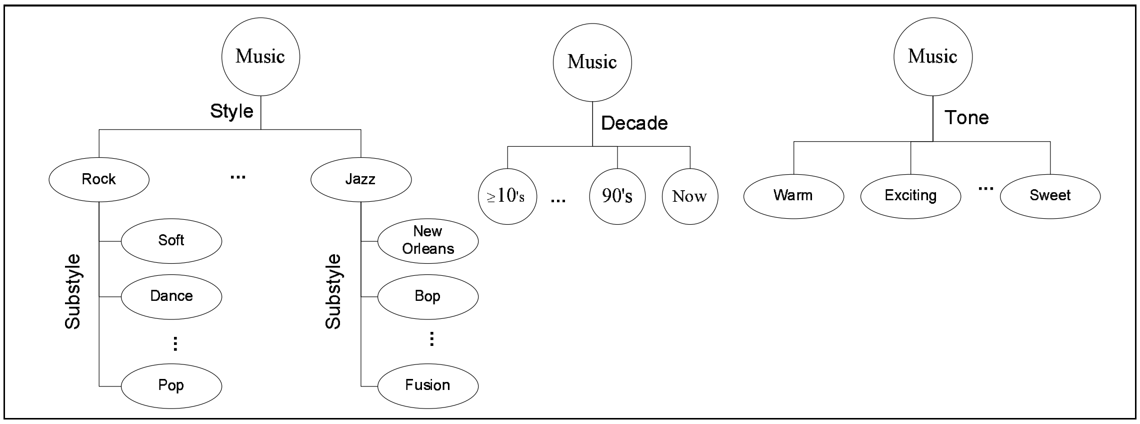

Figure 7.5: Classification Hierarchies

- Classifying queries and documents (Section 7.5): Imprecise classifiers can map too many documents and queries to higher levels of the hierarchy, making searches more expensive. What are the options for building classifiers? Are they precise enough for our needs? What is the impact of classification errors?

- SON membership (Section 7.6): When should a node join a SON? What is the cost of joining a SON? Can we reduce the number of SONs that a node needs to belong to (while being able to find most results)?

- Searching SONs (Section 7.7): How do we search SONs? Is it worth having Semantic Overlay Networks? Is the search performance of a SON-based system better than a single-overlay network system such as Gnutella?

7.4 Classification Hierarchies

In this section we present the challenges and some solutions to the problem of choosing a good classification hierarchy for a SON-based system. Specifically, we will define what a good classification hierarchy is, how we can evaluate a classification hierarchy, and how we can choose among a set of possible hierarchies.

A good classification hierarchy is one that: (i) produces buckets with documents that belong to a small number of nodes, (ii) nodes have documents in a small number of buckets, and (iii) allows for easy-to-implement classification algorithms that make a low number of errors (or no errors at all). In the following paragraphs we explain the rationale behind these criteria.

We need a classification hierarchy that produces buckets of documents that belong to a small number of nodes because the smaller the number of nodes we need to search, the better the query performance. To illustrate, consider a classification hierarchy for a music-sharing system that is based on the decade in which the music piece was originally created. In such a system, we may expect that a large number of nodes will have "90's or current" music. If that is the case, there is little advantage to creating a SON for "90's or current" music, as this SON will have almost all nodes in the system and it will not produce any benefit (but the system will still be incurring the cost of an additional connection at each node, and of having to classify nodes and queries).

We need a classification hierarchy such that nodes have documents in a small number of buckets as each bucket will potentially become a SON that needs to be handled by the node. The greater the number of SONs, the greater the cost for a node to keep track of all of them. For example, consider a classification hierarchy for a music-sharing system that is based on a random hash of the music file. If we assume that nodes have a lot more files than there are hash buckets, then we can expect with a high probability that a node will have to join *all* SONs in the system. In this case, the node will have to process every single query sent into the system, eliminating all the benefits of SONs.

Finally, we want classification hierarchies for which it is possible to implement efficient classifiers that make a small number of errors. To illustrate, consider an image sharing system with a classification hierarchy that includes the concept "has a person smiling." This concept may generate a good number of small SONs, but it requires a very sophisticated classification engine that may generate a large number of erroneous results.

Using the criteria for "goodness" of a classification hierarchy presented above, we can now evaluate classification hierarchies (with the final objective of choosing the best one). This evaluation is a very important step as we have seen that if we are not careful in choosing a good classification hierarchy we may reduce or even eliminate the benefits of using SONs. To evaluate, first, we need to make sure that classifiers can be implemented and that they are efficient. Then, we use the actual data from the nodes in the system to predict the size of the SONs as well as the number of SONs to which a node will belong.

7.4.1 Experiments

To illustrate the issues described in this section, we will now evaluate three classification hierarchies for a music sharing system. While our experimental results in this chapter are particular to this important application, we have no reason to believe they would not apply in other applications with good classification hierarchies.

In Figure 7.5, we illustrate three possible classification hierarchies for music. In the figure we only present a small subset of the concepts in each classification hierarchy. The full sets of concepts are presented in Appendix B and are based on the hierarchy used by *All Music Guide* [93], a music database maintained by volunteers who manually classify songs and artists.

The first classification hierarchy divides music files according first to their style (e.g., Rock, Jazz, Classic, etc.), and then to their substyle (e.g., Soft Rock, Dance Rock, etc.). For style, there are a total of 26 categories and a music file can only belong to one category; while for substyle, there are 255 categories and a file can be classified in multiple substyles. The second hierarchy classifies music files based on the decade in which the piece was originally published (10's or before, 20's, ..., 80's, and 90's or newer). Music files can only be classified in one decade. Finally, the third classification hierarchy divides files according to the "tone" of the piece (e.g., warm, exciting, sweet, energetic, party, etc.). There are a total of 128 tones and a music file can be classified in multiple tones.

In our experiment, we used the crawl of 1800 Napster nodes made at the University of Washington during the month of May 2001 [73]. This crawl included the identity of the node (user name), and for each node, the listing of its files. For most nodes,

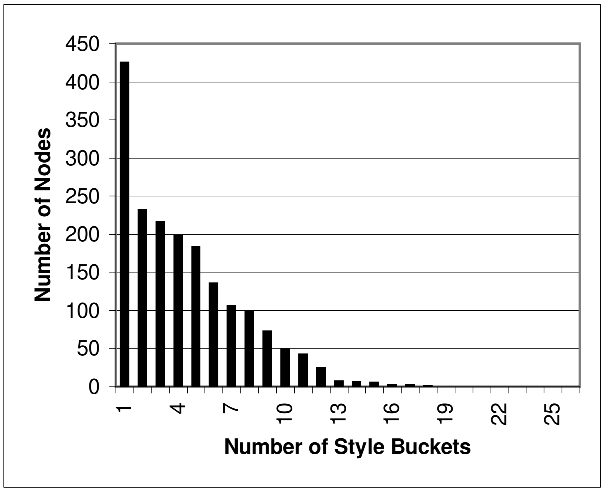

Figure 7.6: Distribution of Style Buckets

filenames were of the form "directory/author-song title.mp3" which allowed us to easily classify files by author and song title. There was additional information (length of file, bit rate, and a signature of the content) that was not used in our evaluations. Actual file content was not available.

To classify documents into the hierarchy, we used the web interface to the database of *All Music Guide* (at allmusic.com). Basically, given a song and artist, the All-Music-Guide database returns the song style, one or more substyles, the decade when the song was released, and one or more tones expressed by the song. We will describe and analyze the classifier in further detail, including how to deal with mistakes and songs that are not in the database, in Section 7.5.1.

To evaluate the style/substyle classification hierarchy, we will first evaluate the style classification hierarchy by itself and then (if needed) we will add to the evaluation the substyle dimension. In Figure 7.6, we show the distribution of Style buckets. To generate this graph, for each node we counted the number of style categories for which the node had one or more files. Then we counted the number of nodes with the same number of style categories and plotted it on the graph. For example, if a node had files in the Rock, Jazz, Country, and Classic styles (and no files in the other styles),

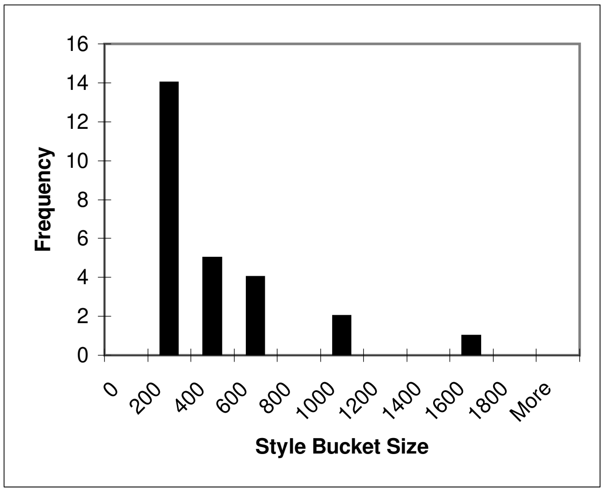

Figure 7.7: Bucket Size Distribution for Style Hierarchy

then the node would have been counted in the bar for "4 style" buckets. From the graph, we can see that 425 nodes (about 24% of the total nodes) have files in just one style. Moreover, 90% of the nodes have files in eight or fewer style categories. This result means that if we define a SON based on the style of files, most nodes will have to handle very few connections.

As indicated before, the smaller the SON, the better query performance will be. However, we cannot compute the size of the Style SONs without the specific node-to-SON assignment strategy. Therefore, we will assume the most conservative strategy: a node will belong to a Style SON if it has one or more files in that Style bucket. Figure 7.7 shows a histogram for the number of nodes that have one or more files in each Style bucket. To generate this graph we counted, for each style, the number of nodes that have one or more files classified in that style. We then counted how many styles had a number of nodes in the ranges 0 to 199, 200 to 399, and so on, and plotted them on the graph. For example, the leftmost bar in the graph means that 14 styles buckets had documents that belonged to between 200 and 399 nodes. The high frequency for bucket size in the interval [200,399] is good news as it shows that the maximum size of most SONs will be small with only 11% to 22% of the nodes.

However, there is one style bucket (shown by the rightmost bar) that has documents belonging to between 1600 and 1800 nodes. Thus, almost all nodes in the system have one or more documents for that bucket (this bucket corresponds to the style "Rock"). Given that there is little advantage in creating a SON based on the style "Rock," we need to explore if it is possible to subdivide it further by using substyles.

We now consider SONs based on the substyle classification. Although the previous analysis showed that we only need to subdivide the Rock style category (and perhaps the two other categories with documents belonging to between 1000 and 1200 nodes), for completeness we will analyze all substyles categories.

In Figure 7.8 we now show the substyle distribution, analogous to Figure 7.6. From the graph, we can see that 328 nodes (about 18% of the total nodes) have files in just one substyle. Moreover, 90% of the nodes have files in 30 or fewer substyle categories. These results are again positive as it shows that the number of SONs to which most nodes may belong is small. In Figure 7.9 we show the bucket size histogram, analogous to Figure 7.7. From the figure we can see that 222 of the substyles (87% of the total) will have documents belonging to less than 400 nodes. However, there are again a few substyle categories that will have documents that belong to a large number of nodes, but this problem is not as bad as the one that we had when using the style classification hierarchy by itself. In particular, the category with the most number of nodes, "Alternative Pop Rock," (which is represented by the rightmost bar in the histogram) will have documents belonging to only 1031 nodes (57% nodes). Even though the "Alternative Pop Rock" SON will have many nodes, it is still half the size of a full Gnutella network that links all the nodes. In conclusion, a combined style and substyle classification hierarchy is a good candidate for defining SONs as the maximum number of SONs that a node needs to join is small and the maximum number of nodes in a SON is also relatively small.

In Figure 7.10 and 7.11 we analyze the usage of Decades as a criteria for classifying documents. From the figures we can see that decade is not a good classification criteria. Although Figure 7.10 shows that most nodes have documents in only a few decade buckets, Figure 7.10 shows that half the SONs will have more than 600 nodes. In fact, almost all nodes will have documents for the 70s, 80s, and 90s buckets.

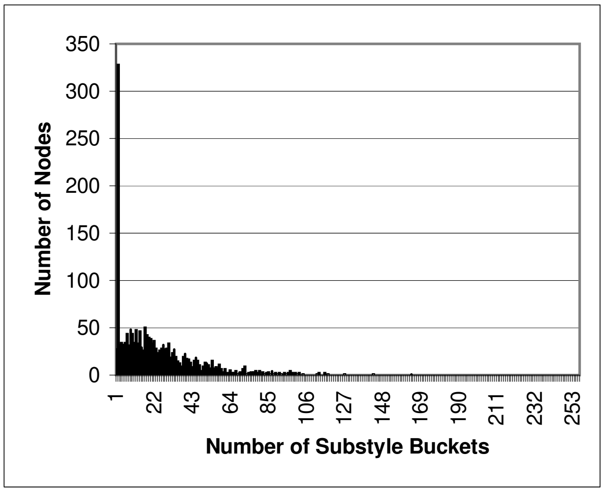

Figure 7.8: Distribution of Substyle Buckets

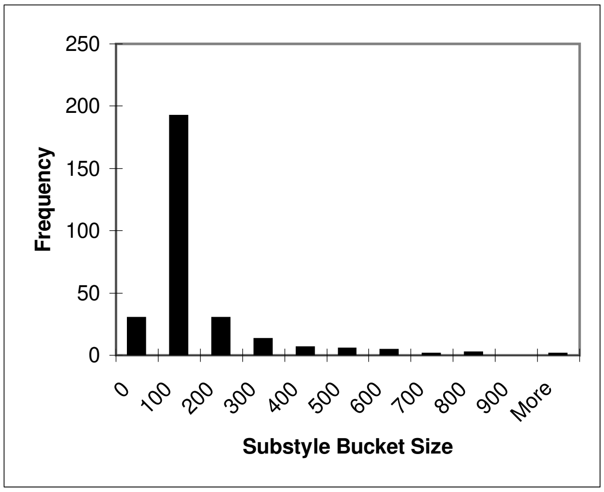

Figure 7.9: Bucket Size Distribution for Substyle Hierarchy

Therefore, given that we do not have a way of subdividing those decades, we have to reject the decade classification hierarchy.

In Figure 7.12, we show the distribution of tone buckets (for an explanation of

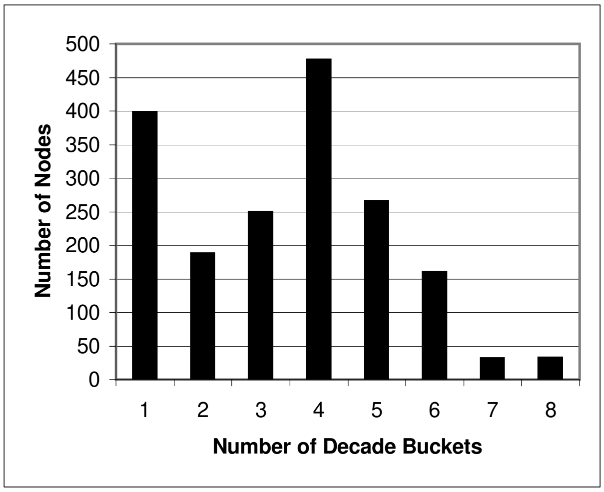

Figure 7.10: Distribution of Decade Buckets

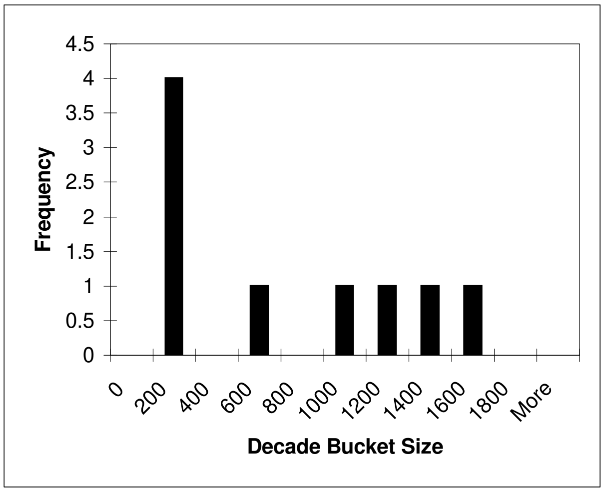

Figure 7.11: Bucket Size Distribution for Decade Hierarchy

the graph, see the description of Figure 7.6). In the graph we can observe that the median number of buckets for which a node has documents is 43, which will result in nodes belonging to a high number of SONs. However, we can see in Figure 7.13 that

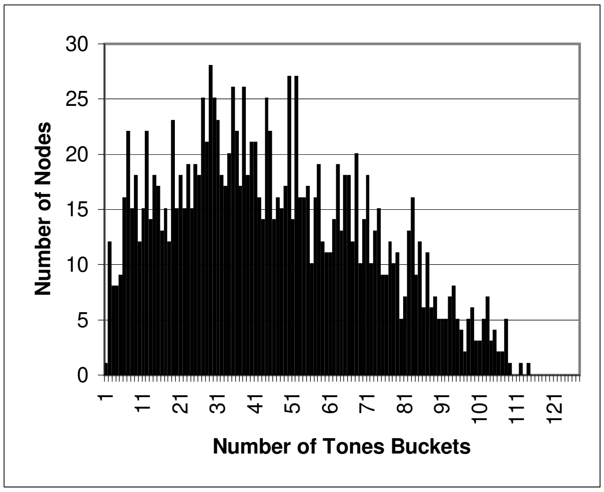

Figure 7.12: Distribution of Tone Buckets

most buckets will contain documents belonging to a relatively small number of nodes. Specifically, 60% of the buckets will have documents belonging to 625 or fewer nodes, and 90% of the buckets will have documents belonging to 875 nodes. In conclusion, using a classification hierarchy based on tone is borderline, and depending on the specifics of the tradeoff between nodes maintaining a large number of connections and the benefits of relatively small SONs, we may decide to use it or not. Nevertheless, of all the classification hierarchies evaluated, the one based on style and substyle is clearly superior and we will use it in the rest of our experiments.

7.5 Classifying Queries and Documents

In this section we describe how documents and queries are classified. Although the problem of classifying documents and the problem of classifying queries are very similar, the *requirements* for the document and query classifiers can be very different. Specifically, it is reasonable to expect that nodes will join a relatively stable P2P network at a low rate (a few per minute); while we could expect a much higher query rate (hundreds or even more per second). Additionally, node classification is more

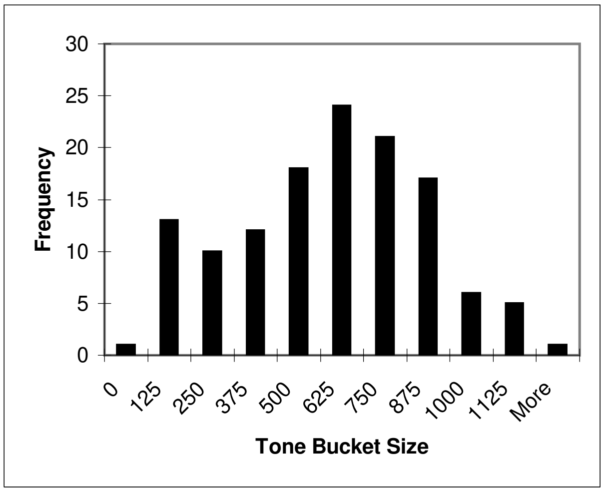

Figure 7.13: Bucket Size Distribution for Tone Hierarchy

bursty, as when a node joins the network it may have hundreds of documents to be classified; on the other hand, queries are likely to arrive at a more regular rate. Under these conditions, the document classifier can use a very precise (but time consuming) algorithm that can process in batch a large number of documents; while, the query classifier must be implemented by a fast algorithm that may have to be imprecise.

The classification of documents and queries can be done automatically, manually, or by a hybrid process. Examples of automatic classifiers include text matching [64], Bayesian networks [67], and clustering algorithms [85]. These automatic techniques have been extensively studied and they are beyond the scope of this dissertation. Manual classification may be achieved by requiring users to tag each query with the style or substyle of the intended results. For example, the user may indicate that results for the query "Yesterday" are expected to be in the "Oldies" substyle; or that results for the query "Like a rolling stone" are expected to be in the "Rock" style. If the user does not know the substyle or style of the potential results, he can always select the root of the hierarchy so all nodes are queried. Similarly, the node manager can also select which SONs his node should join. Finally, hybrid classifiers aid the manual classification with databases as we will see shortly in our experiments.

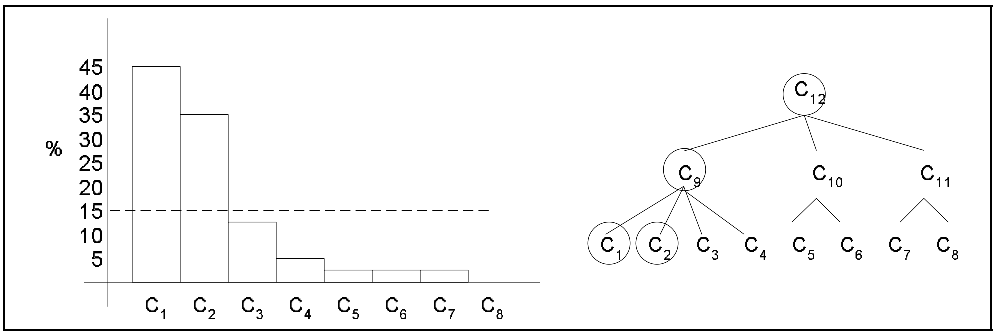

Figure 7.14: Choosing SONs to join

7.5.1 Experiments

The goal of this experimental section is to show that we can classify documents and queries and to study the precision of those classifications.

Evaluating our Document Classifier

Documents were classified by probing the database of All Music Guide at allmusic.com [93]. In this database songs and artists are classified using a hierarchy of style/substyle concepts equivalent to the leftmost classification hierarchy of Figure 7.5. Recall that for each Napster node used in our evaluation we had a list of filenames with the format "directory/author-song title.mp3." As a first step, the document classifier extracted the author and the song title for the file. The classifier then probed the database with that author and song and obtained a list of possible song matches. Finally, the classifier selected the highest rank song and found its style and substyles. If there were not matches in the database, the classifier assigned "unknown" to the style and substyle of the file.

There were many sources of errors when using our document classifier. First, the format of the files may not follow the expected standard, so the extraction of the author and song title may return erroneous values. Second, we assumed that all files were music (but Napster could be, and was actually used, to share other kind of files). Third, users made misspellings in the name of artists and songs (to reduce the effect of misspellings, we used a phonetic search in the All Music database, so some common misspellings did not affect the classification). Finally, the All Music database is not complete, which is especially true in the case of classical music.

To evaluate the document classifier, we measured the number of incorrect classifications that it made. We selected 200 random filenames and manually found the substyles to which they belonged (occasionally using the All Music database and Google as an aid to find the substyles of less known pieces). We then compared the manual classification with the one obtained from our document classifier. We considered a classification to be incorrect for a given document if the document classifier returned one or more substyles to which the document should not belong. Note that an "unknown" classification from our classifier was not considered incorrect as it would correspond to the root node of the classification hierarchy. In our evaluation, we found that 25% of the files were classified incorrectly.

It is important to note that not every misclassified document will not be found later on. To evaluate the true effect of document misclassification, we evaluated the impact of an incorrect document classification on the assignment of nodes to SONs. For this experiment, we selected 20 random nodes, we classified all their documents, and assigned the nodes to all the substyles of their respective documents. We considered a classification to be incorrect for a given *node* if the node was not assigned to one or more substyles to which the node should belong. In our evaluation, we found that only 4% of the nodes were classified incorrectly. This result shows that errors when classifying documents tend to cancel each other within a node. Specifically, even if we fail to classify a document as, for example, "Pop Rock," it is likely that there will be some other "Pop Rock" document in the node that will be classified correctly so the node will still be assigned to the "Pop Rock" SON. Nevertheless, misclassified documents are still a problem for exhaustive queries, however, in practice almost all queries in P2P systems are partial.

Evaluating our Query Classifier

For our experiments, queries were classified by hand. Queries were either classified in one or more substyles, a single style, or as "music"(the root of the hierarchy). In our experiments we used queries obtained from traces of actual queries sent to an

OpenNap server run at Stanford [95]. Thus, by manually classifying queries, we are "guessing" what the users would have selected from, say, a drop-down menu as they submitted their queries.

Unfortunately, we cannot evaluate the correctness of the query classification method (we, of course, consider our manual classification of all queries to be correct). Nevertheless, we can study how precise our manual classification was (i.e., how many times queries were classified into a substyle, a style, or at the root of the classification hierarchy). We selected a trace of 50 *distinct* queries (the original query trace contained many duplicates which the authors of [95] believed were the result of cycles in the OpenNap overlay network) and then manually classified those queries. The result was that 8% of the queries were classified at the root of the hierarchy, 78% were classified a the style level of the hierarchy and 14% at the substyle level. As we will see in Section 7.7, the distribution of queries over hierarchy levels will impact the overall system performance, as more precisely classified queries can be executed more efficiently.

7.6 Nodes and SON Membership

In Section 7.3 we presented a conservative strategy for nodes to decide which SONs to join. Basically, under this strategy, nodes join all SONs associated with a concept for which they have a document. This strategy guarantees that we will be able to find all the results, but it may increase both the number of nodes in each SON and the number of connections that a node needs to maintain. A less conservative strategy, where nodes join some of all the possible SONs, can have better performance. In the next subsection we introduce a non-conservative assignment strategy: Layered SONs.

7.6.1 Layered SONs

The Layered SONs approach exploits the very common zipfian data distribution in document storage systems. (It has been shown that the number of documents in a website when ranked in order of decreasing frequency, tends to be distributed according to Zipf's Law [40].) For example, on the left side of Figure 7.14 we present a hypothetical histogram for a node with a Zipfian data distribution (we'll explain the rest of the figure shortly). In this histogram we can observe that 45% of the documents in the node belong to category c_1, about 35% of the documents belong to category c_2, while the remaining documents belong to categories c_3 to c_8. Thus, which SONs should the node join? The conservative strategy mandates that the node needs to join SON_{c_1} through SON_{c_8}. However, if we assume that queries are uniform over all the documents in a category, it is clear that the node will have a higher probability of answering queries in SON_{c_1} and SON_{c_2} than queries in the other SONs. In other words, the benefit of having the node belong to SON_{c_1} and SON_{c_2} is high, while the benefit of joining the other SONs will be very small (and even negative due to the overhead of SONs). A very simple and aggressive alternative would be to have the node join only SON_{c_1} and SON_{c_2}. However, this alternative would prevent the system from finding the documents in the node that do not belong to categories c_1 and c_2.

We propose Layered SONs, an approach where nodes determine which SONs to join based on the number of documents in each category. To illustrate, consider again Figure 7.14. At the right of the figure we present the hierarchy of concepts that will aid a node in deciding which SONs to join. In addition, a parameter of the Layered SON approach is the minimum percentage of documents that a node should have in a category to belong to the associated SON (alternatively, we can also use an absolute number of documents instead of a percentage). In the example, we have set that number at 15%. Let us now determine which SONs the node with the histogram at the left of Figure 7.14 should join. First, we consider all the base categories in the hierarchy tree (c_1 to c_8). As c_1 and c_2 are above 15%, the node joins SON_{c_1} and SON_{c_2}. As all the remaining categories are all below 15%, the node does not join their SONs. We then consider the second level categories (c_9, c_{10}, and c_{11}). As the combination of the non-assigned descendants of c_9, c_3 and c_4, is higher than 15%, the node joins SON_{c_9}. However, the node does not join the SON of c_{10} as the combination of c_5 and c_6 are not above 15%. Similarly the node does not join the SONs of c_{11} as

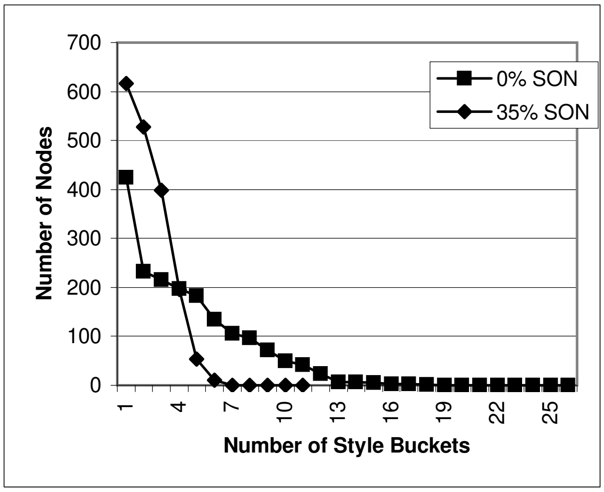

Figure 7.15: Distribution of Style SONs

c_7 and c_8 are below the threshold. Finally, the node joins the SON associated with the root of the tree ($SON_{c_{12}}$) as there were categories (c_5, c_6, c_7 and c_8) that were not part of any assignment. This final assignment is done regardless of the 15% threshold as this ensures that all documents in the node can be found (in our example, if we do not join $SON_{c_{12}}$ we will not be able to find the documents in the SONs of c_5, c_6, c_7 and c_8).

Note that the conservative assignment is equivalent to a Layered SON where the threshold for joining a SON has been set to 0%. In this case, the node will join the SONs associated with all the base concepts for which it has one or more documents.

7.6.2 Experiments

In this subsection we contrast the result, in terms of SON size and number of SONs per node, of the conservative approach of Section 7.4.1 and the Layered SON approach. As the Style/Substyle classification hierarchy was the clear winner in Section 7.4.1, in this section we will only consider it and we will not analyze the other classification hierarchies.

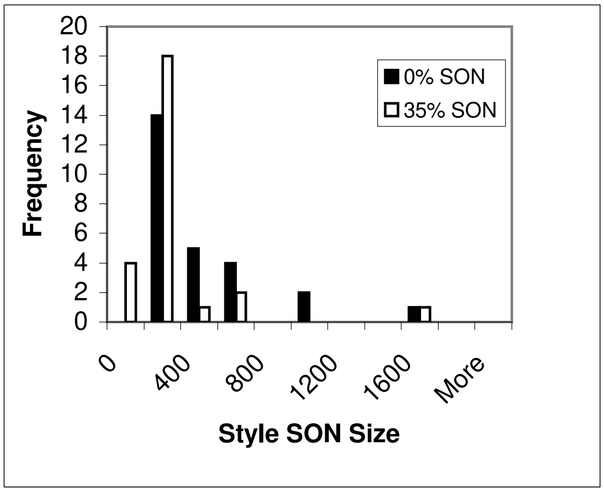

Figure 7.16: SON Size Distribution for Style Hierarchy

In Figure 7.15, we show the distribution of style SONs when using Layered SONs with a threshold of 35% and for the conservative assignment (labeled as 0% SON). The graphs do not include the "root" category to which, in practice, all nodes belong. From the graph, we can see that 616 nodes (about 34% of the total nodes) need to belong to just one style. This result shows a significant improvement versus the conservative assignment of Section 7.4.1 when only 24% of the nodes belonged to one style. Moreover, 97% of the nodes need to belong to four or less style categories (versus 90% when doing conservative assignments).

Using layered SONs also helps reduce the number of nodes per SON. Figure 7.16 shows a histogram for the size of the SONs (excluding the "root" SON). From the graph we can see that by using Layered SONs we have a larger number of small SONs. However, as before, we still have a problem with the "Rock" style (rightmost bar in the graph) to which almost all nodes will have to belong. In conclusion, there is a significant reduction in the size of SONs when using Layered SONs instead of the conservative strategy. This reduction will lead to significant improvements in query performance.

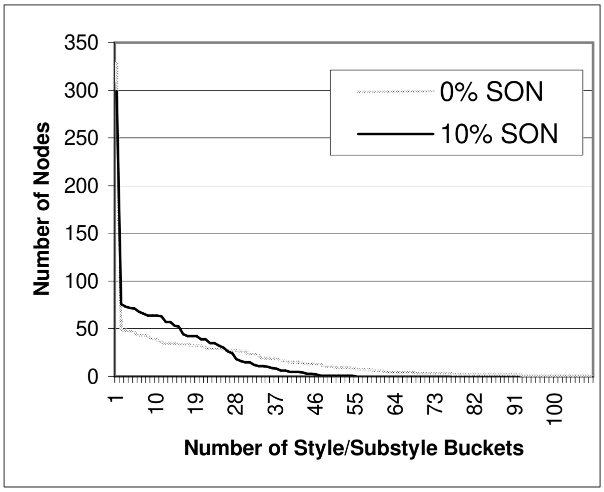

Figure 7.17: Distribution of Substyle SONs

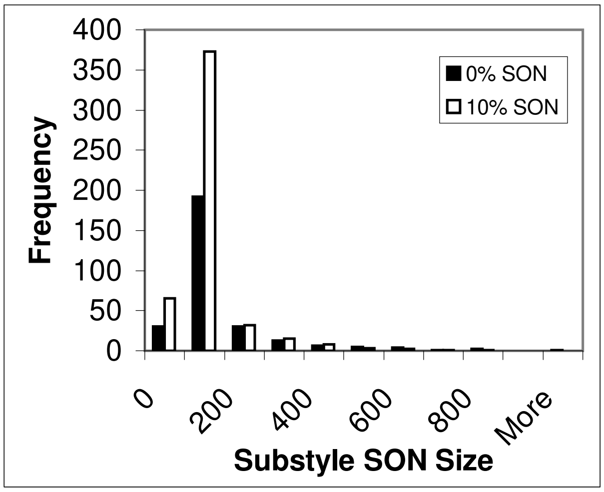

Figure 7.18: SON Size Distribution for Substyle Hierarchy

We now consider Layered SONs based on the Style/Substyle classification hierarchy. In Figure 7.17 we show the size distribution of Style/Substyle SONs for Layered SONs with a threshold of 10% and for the conservative assignment (labeled as 0%

SON). From the graph, we can see that the Style/Substyle Layered SON and the conservative assignment strategy behave similarly in terms of the number of connections required at each node. However, the advantage of Layered SONs can be seen in Figure 7.18 which shows the size histogram of SONs. When using Layered SONs, SONs will have on average 135 nodes (versus 517 nodes for the conservative approach). Moreover, the Layered SON does not have any SONs with more than 875 nodes, while the conservative approach has 24. In conclusion, using Layered SONs with a Style/Substyle hierarchy produces a significant improvement versus the conservative assignment as we have much smaller SONs.

7.7 Searching SONs

As explained in Section 7.3, queries can be exhaustive or partial. In the case of an exhaustive query, we need to find all matches, so all SONs that may contain results need to be considered; while in the case of partial queries we do not need to consider all of them. In this section, we explore the problem of how to choose among a set of SONs when using Layered SONs.

7.7.1 Searching with Layered SONs

Searches in Layered SONs are done by first classifying the query. Then, the query is sent to the SON (or SONs) associated with the base concept (or concepts) of the query classification. Finally, the query is progressively sent higher up in the hierarchy until enough results are found. In case more than one concept is returned by the classifier, we perform a sequential search in all the concepts returned before going higher up in the hierarchy. For example, when looking for a “Soft Rock” file we start with the nodes in the “Soft Rock” SON. If not enough results are found (recall that partial queries have a target number of results), we send the query to the “Rock” SON. Finally, if we still have not found enough results, we send the query to the “Music” SON. Note that there are multiple approaches when searching with Layered SONs. In this chapter we are concentrating on a single serial one (as our objective is to minimize number of messages). However, there are other approaches such as searching more than one SON in parallel (by asking each one for some fraction of the target results) which may result in a higher number of messages, but will start producing results faster.

Note that this search algorithm does not guarantee that all documents will be found if there are classification mistakes for documents. Not finding all documents may or may not be a problem depending on the P2P system, but in general, if we need to find all documents for a query (in the presence of classification mistakes), our only option is an exhaustive search among all nodes in the network. However, we will see that with our document classifier (which has an per-document classification mistake probability of 25%), we can find more than 95% of the documents that match a query. In addition, this search algorithm may result in duplicate results. Specifically, duplication can happen when a node belongs, at the same time, to a SON associated with a substyle and to the SON associated with the parent style of that substyle. In this case, a query that is sent to both SONs will search the node twice and thus it will find duplicate results.

7.7.2 Experiments

We will now consider two possible SON configurations and evaluate their performance against a Gnutella-like system. As before, we used the crawl of 1800 Napster nodes made at the University of Washington, which were classified using the All Music database. We assumed that the nodes in the network (both inside SONs and in the Gnutella network) were connected via an acyclic graph and that on average each node was connected to four other nodes. Although the assumption of an acyclic graph is not realistic, we are considering acyclic networks as the effect of cycles is independent of the creation of SONs. Cycles affect a P2P system by creating repeated messages containing queries that the receiving nodes have already seen. Therefore, an analysis of an acyclic P2P network gives us a lower estimate of the number of messages generated.

To illustrate, we will first show the result for a single query when using a Layered

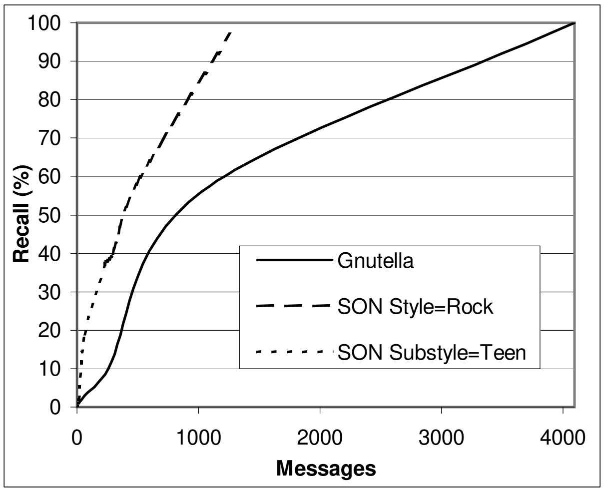

Figure 7.19: Number of messages for query "Spears"

SON for the style/substyle classification hierarchy. In Figure 7.19 we evaluate the performance of the query "Spears" (classified manually as a "teen-pop"). The figure shows the level of recall versus the number of messages transmitted. The level of recall is the ratio between the number of matches obtained versus the number of matches that would be obtained if we searched all nodes in the system. The data points in the graph were obtained by averaging 50 simulations over randomly generated network topologies. As indicated before, when using Layered SONs, we may obtain duplicate matches. In such a case, we did not count duplicate results as new matches. Following the search algorithm for Layered SONs, the query was initially sent to the "Teen Pop" SON. We show as a dotted line in the graph the recall level versus message performance of that SON. After the "Teen pop" SON is searched (consuming 232 messages and yielding 37% of the matching documents), the system searches the parent of "Teen pop", i.e., the "Rock" SON. We show the recall level versus message performance of this next SON as a dashed line. Finally, we show as a solid line the recall level versus message performance of a Gnutella-like system that searches all nodes (in an order that is independent from the content). From the graph, we can see that the Layered SON setup is able to find results with significantly fewer messages

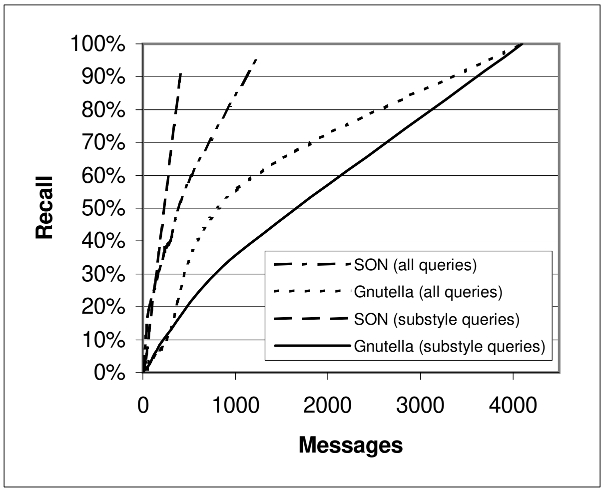

Figure 7.20: Average number of Messages for a Query Trace

(and therefore much faster) than the Gnutella network. Specifically, the SON-base system was able to find 20% of the results with only 92 messages, while it took 285 messages for the Gnutella system to reach that same level.

As an additional observation note that the Layered SON system does not find all results available. While Gnutella finds 100% of the results in the system, Layered SON only found 97% of the results. The reason is that the document classifier made some mistakes and some nodes (with Spears documents) were not assigned to the "Rock" or "Teen Pop" SONs. If we would like to find the remaining 3% of the documents, we would have to send the query to the "Music" SON (which contains all nodes). Such an action would have the same cost as a standard Gnutella search (plus the overhead of having searched in the "Teen Pop" and "Rock" SONs before). Of course in practice most users will never want to perform an exhaustive search [11].

Let us now analyze the performance of Layered SONs with a stream of queries. For this experiment we used 50 different random queries obtained from traces of actual queries sent to an OpenNap server run at Stanford [95]. These queries were classified by hand as described in Section 7.6. Queries classified at the substyle level were sent sequentially to the corresponding SON (or SONs), and then to the style-level SON.

Queries classified at the style level, were first sent sequentially to all substyles of that style, and then to the style level. Queries classified at the root of the hierarchy were sent to all nodes. We measured the level of recall, averaged for all 50 queries, versus the number of messages sent in the system. As in the previous experiment, the graphs were obtained by running 50 simulations over randomly generated network topologies.

In Figure 7.20, we show the result of this experiment. The figure shows the number of messages sent versus the level of recall. As with the case of a single query, Layered SONs were able to obtain the same level of matches with significantly fewer messages than the Gnutella-like system. Again, Layered SONs do not achieve recall levels of 100% in general (average maximum recall was 93%) due to mistakes in the classification of nodes.

The results of Figure 7.20 show the average performance for all query types (dotted line). However, if a user is able to precisely classify his query, he will get significantly better performance. To illustrate this point, Figure 7.20 also shows with a dashed the number of messages sent versus the level of recall for queries classified at the substyle level (the lowest level of the hierarchy). In this case, we obtain a significant improvement versus Gnutella. For example, to obtain a recall level of 50%, Layered SONs required only 461 messages, while Gnutella needed 1731 messages, a reduction of 375% in the number of messages. Moreover, even at high recall levels, Layered SONs were able to reach a recall level of 92% with about 1/5 of the messages that Gnutella required.

Note that the shape of the curve for the message performance of Gnutella is slightly different for all queries and for queries classified at the substyle level. The reason for this difference is very subtle. We were only able to classify very precisely (i.e. to the substyle level) queries for songs that are very well known. Due to their popularity, there are many copies of these songs throughout the network. Therefore, a Gnutella search approach will have a high probability of finding a match in many of the nodes visited, making the flooding of the network less of a problem than with more rare songs. Nevertheless, even in this case, Layered SONs performed much better than Gnutella.

7.8 Discussion

In this chapter we studied how to improve the efficiency of a peer-to-peer system, and in particular of a federation of ARs, by clustering nodes with similar content in Semantic Overlay Networks (SONs). We showed how SONs can efficiently process queries while preserving a high degree of node autonomy. We introduced Layered SONs, an approach that improves query performance even more at a cost of a slight reduction in the maximum achievable recall level. From our experiments we conclude that SONs offer significant improvements versus random overlay networks, while keeping costs low.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Many traditional library materials, such as preprints, scientific data, scholarly journals, and even books, have moved on-line. This corpus is a wealth of digital information that is worth preserving. The trend of moving library information to electronic media is likely to continue in the future as on-line access has become the preferred means of keeping up with new research in many fields [22]. Therefore, preserving electronic information will only become more critical as more and more information is created only in digital format.

Electronic information is inherently perishable: if we do nothing, it will not be preserved by accident. The fragility of digital media goes well beyond that of paper. While paper can last for hundreds of years (if it is of good quality and has the proper environment), digital documents can only be accessed at best for a few decades, unless we take action. The fragility of digital documents is the result of the continuing evolution of the components (such as technologies and formats) necessary to access them. We have already witnessed to technologies and formats that seemed to be ubiquitous being replaced by newer, and frequently incompatible, substitutes. To illustrate, consider the 5 1/4 inch floppy disk; omnipresent for 15 years, it has been completely replaced by 3 1/2 inch disks. This replacement of technologies and formats is continuing. In fact, JPEG, the standard for images that is used by most digital cameras, is in the process of being replaced by JPEG 2000, a higher quality, higher compression, image format. If JPEG is rendered obsolete by JPEG 2000, in just a few years, it will be difficult for users to look back and see the pictures that they are taking today [82].

In this dissertation, we addressed the problem of data preservation by building a reliable archival repository that protects digital information from failures. The repository takes actions to protect the documents through participation in a Peer-to-Peer federation of repositories that store copies of each others' data. In addition, we designed the repositories so that they could support efficient search and access of their content. This dissertation was divided into three parts: (i) design and evaluation of archival repositories, (ii) a reference architecture and algorithms for ARs, and (iii) efficient access to a federation of ARs.

In the first part of this dissertation, we described in detail the design and evaluation of Archival Repositories. First, we presented a comprehensive model for an AR, including options for the most common recovery and preventive maintenance techniques. Then, we introduced ArchSim, a powerful simulation tool for evaluating ARs and for studying available archival strategies. We then used the model and ArchSim to analyze a hypothetical Technical Report repository operated between two universities. Through this analysis, we evaluated how AR factors, such as disk reliability, handling of format failures, and preventive maintenance, affect the reliability of the system. In addition to reliability, the most important factor in evaluating an AR is cost. Therefore, we also studied in detail how to make cost-driven decisions about archival repositories.

In the second part of this dissertation, we presented a reference architecture and algorithms for ARs. In particular, we presented the *Cellular Repository Architecture*, an architecture for long-term archival storage of digital objects. This architecture allows the construction of a simple, yet powerful, archival repository by using signatures as object handles, not allowing deletions, having awareness services in all layers, and using only disposable auxiliary structures. We argued that this architecture is well suited for a heterogeneous and evolving environment because each site needs only to agree on some very simple interfaces, a signature computation function, and some simple object header structures.

We also studied the algorithms for awareness services. These algorithms are fundamental for our Cellular Repository Architecture as they are present at every layer. The main contribution of this study was a unified view of the different awareness algorithms. Specifically, we presented awareness schemes as variants of two unifying algorithms (UNI-AWARE and DIST-UNI-AWARE). This approach made differences in assumptions and performance of the different algorithms very apparent. Furthermore, such a unified view of awareness mechanisms could be extremely important for a client that must deal with stores that implement different schemes.

In the third part of this dissertation, we focused on how to provide efficient searches in a massive peer-to-peer federation of nodes. We presented two ways of achieving better performance: Routing Indices and Semantic Overlay Networks. Routing Indices achieve greater efficiency by placing compact indices at each node. We presented three RI schemes: the compound, the hop-count, and the exponential routing indices, and evaluated their performance via simulations. We found that RIs can improve performance by one or two orders of magnitude vs. a flooding-based system, and by up to 100% vs. a random forwarding system.

Semantic Overlay Networks improve the efficiency of a peer-to-peer system by clustering nodes with similar content in the same overlay network. We showed how SONs can efficiently process queries while preserving a high degree of node autonomy. We introduced Layered SONs, an approach that improves query performance even more at the cost of a slight reduction in the maximum achievable recall level. It is important to notice that the results in the third part of this dissertation can help improve the search performance not only of ARs but also of current and future P2P systems.

8.2 Future Work

Many important issues related to long-term archiving of information remain unexplored in this dissertation. Interesting questions can arise in the design and implementation of an Archival Repository based on the techniques presented here. Below, we discuss such issues, and outline directions for future research.

Preserving the meaning: Our focus in this dissertation was data preservation (preservation of bits). Although preserving the bits is a fundamental and necessary step for meaning preservation, without a way of preserving the meaning, digital documents would not be not fully protected. Three solutions have been proposed for the problem of meaning preservation: *migration,* which consists of transcribing documents from old formats into new formats; *emulation*, that allows applications built for obsolete hardware to run in new machines; and *encapsulation*, that wraps the digital document with metadata that allows the recreation of the hardware and software necessary to understand the document. However, all these solutions are themselves vulnerable to obsolescence and, to ensure long-term preservation, require periodic "fixes," such as, for example, building an emulator for the now old machine for which an emulator of an older machine was built. As we did with data preservation the study of the design issues and cost implications of these (and other) meaning preservation techniques are a necessary step for their successful deployment. To do this study we can use the framework and design techniques presented in this dissertation, but new parameters and an obsolescence model are needed, making meaning preservation an open area of research.

Intellectual Property: In our architecture, digital objects will be served beyond the organization that runs the repository or that owns the information. This means that the repository must understand and enforce intellectual property laws, and must offer its clients a variety of access and payment options. At the same time, any intellectual property mechanism must allow access to the document in the case of disappearance of the holder of the intellectual property rights. Two possible (and complementary) approaches to enforce intellectual property rights are access control and digital marking. Access control makes use of encryption and authentication to ensure that digital objects are provided only to trusted participants. Digital marking consists of watermaking or fingerprinting documents, so the source of a copy can be determined; thus, if the copy were illegal, the source of it could be prosecuted.

Implementation of a large-scale Testbed System: A full deployment of a testbed AR system would allow the checking of parameter values used in the models presented in this dissertation. The results from a testbed system would be invaluable, especially for the measurement of the short and medium-term archival capability and cost of the system. However, researchers and developers of testbed systems need to recognize that a testbed system cannot accurately measure the *long-term* attributes of the archival system as they would need to run for so long that the results obtained from them would be useless.

Replication Techniques: In our work, replication was achieved by creating full copies of the documents. As a result, given a certain level of archival guarantee, we frequently need to round up the number of copies needed to ensure the given guarantee. This rounding up may result in generating more copies than needed. For example, consider 3 sites, each with a 1/10 probability of failing. If we make a copy of a document in two sites, the probability of not losing the document (due to a failure of both sites) is 1/100. If we make a copy in all three sites, then the probability of not losing the documents reduces to 1/1000. Therefore, if we need an archival guarantee in the range $(1/100, 1/1000]$, we would have to make 3 copies. To avoid rounding up, instead of creating full copies, we can create "partial copies," i.e., copies where more than one site is required to re-create the document. Specifically, we could divide the document in 3 pieces and place 2 pieces in each of the three sites in a way that we could always reconstruct the document if 2 sites were available. In this case, we would need 2/3 of the space required by the full copies with a probability of failure of 0.003. In the literature there are many ways to generate partial copies (e.g., dividing the documents or using xor). However, the tradeoffs and implications on archival systems have not been studied.

Untrusted Federation of Local Repositories: For the purpose of this dissertation, we adopted an optimistic view where participants do not purposely try to "sabotage" the system (e.g., destroy all copies of a specific document). Although this assumption is not unrealistic when considering a federation of Digital Libraries, it is definitely not realistic in a less controlled environment. Although some research done in the area of Peer-to-Peer networks is applicable to this problem [19], more work is needed. Possible research directions include: reputation systems, where sites are ranked (and trusted) according to the good or bad behavior that they have shown in the past; authentication, where only trusted participants are allowed into the network; encryption and digital signatures, so a node can determine if messages and documents come from trusted nodes; and "imprecise" algorithms, such as the ones used in LOCKS [66], to prevent outsiders from easily determining the identity of all nodes or all the locations of a document.

Bibliography

[1] M.E. Adiba and B.G. Lindsay. Database snapshots. In *Proceedings of the International Conference on Very Large Databases*, 1980.

[2] Thomas Antognini and Walter Antognini. A flexibly configurable 2d bar code. In *Information Based Indicia Program Technology Symposium*, 1996.

[3] William Y. Arms. Key concepts in the architecture of the digital library. *D-Lib Magazine*, July 1995.

[4] W. David Kelton Averill M. Law. *Simulation Modeling & Analysis*. McGraw-Hill, 1991.

[5] BackWeb. *BackWeb White Paper*.

[6] D. Barbara and R.J. Lipton. A class of randomized strategies for low-cost comparison of file copies. In *IEEE Transactions on Parallel and Distributed Systems*, volume 2(2), pages 160–170. IEEE, April 1991.

[7] R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[8] John W.C. Van Bogart. *NML Storage Technology Assessment: Final Report*. National Media Lab, 1994.

[9] John W.C. Van Bogart. *Magnetic Tape Storage and Handling*. National Media Lab, 1995.

[10] Marthyn Borghuis, Hans Brinckman, Albert Fischer, Karen Hunter, Eleonore van der Loo, Rob ter Mors, Paul Mostert, and Jaco Zijlstra. *TULIP. Final Report.* Elsevier Science, July 1996.

[11] S. Brin. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference*, 1998.

[12] E. Çinlar. *Introduction to Stochastic Processes.* Prentice-Hall, 1975.

[13] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[14] Clip2.com, world-wide web: http://dss.clip2.com/gnutella.html. *Gnutella: To the Bandwidth and Beyond.*

[15] The Commission on Preservation and Access, and The Research Libraries Group. *Report of the Task Force on Archiving of Digital Information*, May 1996.

[16] Compaq. Intellisafe. Technical Report SSF-8035, Smal Form Committee, January 1995.

[17] Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive, 1999. Submitted for publication to ACM DL 2000.

[18] Corporation for National Research Initiatives. *Computer Science Technical Reports (CS-TR).* At http://www.cnri.reston.va.us/home/cstr.html.

[19] Neil Daswani, Hector Garcia-Molina, and Beverly Yang. Open problems in data-sharing peer-to-peer systems. In *Proceedings of the International Conference on Database Theory*, January 2003.

[20] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22(1), January 1988.

[21] Document Management Alliance. *DMA 1.0 Specification Draft*, January 1998.

[22] Margaret Hedstrom et al. Report on the nfs workshop on research challenges in digital archiving: Towards a national infrastructure for long-term preservation of digital information. Technical report, National Science Foundation, 2002.

[23] M. Faloutsos, P. Faloutsos, and C Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, 1999.

[24] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[25] Freenet. *At http://freenet.sourceforge.com.*

[26] Gartner Consulting. *TCO Analyst. A White Paperon GartnerGroupś Next Generation Total Cost of Ownership Methodology*, 1997.

[27] P. Gawen. Computer output to optical disk and its application. In *Proceedings of the Seventh Annual Conference on Optical Information Systems*, pages 102–106, July 1990.

[28] Andrew Goldberg and Peter Yianilos. Towards an archival intermemory. In *Advances in Digital Libraries*, 1998.

[29] G. Gordon. *The Application of GPSS V to discrete System Simulation*. Prentice-Hall, 1975.

[30] L. Gravano and H. Garcia-Molina. Generalizing gloss for vector-space databases and broker hierarchies. In *Proceedings of VLDB*, 1995.

[31] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of gloss for the text-database discovery problem. In *Proceedings of SIGMOD*, 1994.

[32] L. Gravano, H. Garcia-Molina, and A. Tomasic. Precision and recall of gloss estimators for database discovery. In *Proceedings of PDIS*, 1994.

[33] L. Gravano, H. Garcia-Molina, and A. Tomasic. Gloss: Text-source discovery over the internet. *TODS*, 2000.

[34] Joseph Halpern and Carl Lagoze. The Computing Research Repository: Promoting the rapid dissemination and archiving of computer science research. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, August 1999.

[35] R.A. Howard. The science of decision-making. In *Readings on the Principles and Applications of Decision Analysis*, volume 1, 1964.

[36] R.A. Howard. Decision analysis: Practice and promise. In *Management Science*, volume 34, 1988.

[37] Andreas Reuter Jim Gray. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publisher, 1993.

[38] Robert Kahn and Robert Wilensky. A framework for distributed digital object services. Technical Report tn95-01, CNRI, May 1995.

[39] W.H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *Computing Surveys*, 13(2), June 1981.

[40] R Korfhage. *Information storage and retrieval.* Wiley Computer Publishing, 1997.

[41] D. Kossman. The state of the art in distributed queyr processing. *ACM Computing Survey*, September 2000.

[42] Kranch. Preserving electronic documents. In *ACM Digital Library Conference*, 1998.

[43] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[44] B.A. Huberman L.A. Adamic A.R. Puniyani, R. M. Lukose. Search in power-law networks. Technical report, HP Labs, 2001. Available at http://www.hpl.hp.com/shl/papers/plsearch/.

[45] Wilburt Juan Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB*, 1996.

[46] Carl Lagoze, Clifford A. Lynch, and Ron Daniel Jr. The Warwick framework: A container architecture for aggregating sets of metadata. Technical Report TR96-1593, Cornell University, June 1996.

[47] G. G. Langdon. A note on the ziv-lempel model for compressing individual sequences. In *IEEE Transactions on Information Theory*, volume 29(2), pages 284–287, 1983.

[48] Edward K. Lee. Highly-available, scalable network storage. *COMPCON*, 1995.

[49] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. *ASPLOS*, 1995.

[50] Haim Levy. *Stochastic Dominance: Investment Decision Making under Uncertainty*. Kluwer Academic Publishers, 1998.

[51] B.G. Lindsay and et al. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD Annual Conference*, 1986.

[52] Stephen Manes. Time and technology threaten digital archives. *The New York Times*, April 1997.

[53] C. Manning and H. Schutze. *Foundations of statistical natural language processing*. The MIT Press, 1999.

[54] MARIMBA. *Castanet*. At http://www.marimba.com.

[55] J.J. Metzner. Efficient replicated remote file comparison. In *IEEE Transactions on Computers*, volume 40(5), pages 651–660. IEEE, May 1991.

[56] C. K. Miller. *Multicast Networking and Applications*. Addison Wesley, 1998.

[57] W. Nejdl, W. Siberski, M. Wolpers, and C. Schmnitz. Routing and clustering in schema-based super peer networks. In *Submitted to the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[58] Open Software Associates. *NetDeploy*. World Wide Web at http://www.osa.com/products/netdploy/.

[59] T.I. Oren. Application of system theoretic concepts to the simulation of large scale adaptive systems. In *Proceedings of the 6th Hawaii International Conference on Systems Sciences*, pages 435–437, 1973.

[60] Erik Ottem and Judy Plummer. Playing it S.M.A.R.T.: The emergence of reliability prediction technology. Technical report, Seagate Technology Paper, June 1995.

[61] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *GLOBECOM*, 2000.

[62] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of markov chain decision processes. *Mathematics of Operation Research*, 12(3):441–450, 1987.

[63] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall PTR, 1995.

[64] B. Ribeiro-Neto R. Baeza-Yates. *Modern Information Retrieval*. Addison Wesley, 1999.

[65] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, August 2001.

[66] Vicky Reich. Lots of copies keep stuff safe as a cooperartive archiving solution for e-journals. *Science and Technology Librarianship*, Fall 2002.

[67] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill Inc., 1991.

[68] Jr. Ron Daniel and Carl Lagoze. Extending the warwick framework: From metadata containers to active digital objects. *D-Lib Magazine*, November 1997.

[69] Jeff Rothenberg. Ensuring the longevity of digital information. *Scientific American*, 272(1):24–29, January 1995.

[70] Jeff Rothenberg. Avoiding technological quicksand: Finding a viable technical foundation for digital preservation. Technical report, Council on Library and Information Resources (CLIR), Washington DC, 1999.

[71] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[72] M. Sahami and S. Yusufali M.Q.W. Baldonado. Sonia: A service for organizing networked information autonomously. In *Proceedings of the Third ACM Conference on Digital Libraries*, 1998.

[73] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, University of Washington, 2002.

[74] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems*. Digital Press, 1992.

[75] A. Silberschtaz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. Mc Graw Hill, 1996.

[76] Marc Smith and Peter Kollock, editors. *Communities in Cyberspace*. Routledge, 1998.

[77] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[78] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.

[79] A. Tanenbaum and A. Woodhull. *Operating Systems Design and Implementation*. Prentice-Hall, Inc., 1999.

[80] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[81] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. *SOSP*, 1997.

[82] Claire Tristram. Data extinction. *Technology Review*, pages 36–42, October 2002.

[83] C. Stael von Holstein. Assessment and evaluation of subjective probability distributions. Technical report, Economic Research Institute, Stockholm School of Economics, 1970.

[84] Stuart Weibel, Jean Godby, and Eric Miller Ron Daniel. Oclc/ncsa metadata workshop report, March 1995.

[85] I. Witten and E. Frank. *Data Mining*. Morgan Kaufmann Publishers, 1999.

[86] World-wide web: http://communities.msn.com. *Microsoft Network Communities*.

[87] World-wide web: http://gnutella.wego.com. *Gnutella Development Page*.

[88] World-wide web: http://groups.yahoo.com. *Yahoo Groups Home Page*.

[89] World-wide web: http://jxta.org. *JXTA Project*.

[90] World-wide web: http://opennap.sourceforge.net. *OpenNap Home Page*.

[91] World-wide web: http://setiathome.ssl.berkeley.edu. *Seti At Home Page*.

[92] World-wide web: http://vivisimo.com. *Vivisimo Home Page*.

[93] World-wide web: http://www.allmusic.com. *All Music Guide*.

[94] World-wide web: http://www.napster.com. *Napster Home Page*.

[95] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *VLDB*, 2001.

[96] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS*, 2002.

[97] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.

Appendix A

MIT/Stanford CSTR Scenario Parameters

- **AR Description**
  - Initial collection: num_{doc} documents. Each document, d, will have the following four materializations:
    * $\langle d, MIT, disk_i, form_j \rangle$,
    * $\langle d, MIT, disk_k, form_l \rangle$,
    * $\langle d, Stanford, disk_x, form_j \rangle$,
    * $\langle d, Stanford, disk_y, form_l \rangle$.

    Where MIT and $Stanford$ are the two sites; $disk_i$, $disk_k$, $disk_x$, and $disk_y$ are different storage devices; and, $form_j$ and $form_l$ are two different formats.
  - Number of components and types: n_{sto} storage devices, n_{form} formats, 2 sites.
  - Failure dependency graph: $site \rightarrow disk$, when the disk is in the given site.
- **Distributions**
  - Disk Failure distribution during access: $U(1/\phi_{sto})$
  - Format Failure distribution during access: $U(1/\phi_{form})$
  - Site Failure distribution during access: $U(1/\phi_{site})$
  - Disk Failure distribution during archival: $U(1/\phi_{sto})$
  - Format Failure distribution during archival: $U(1/\phi_{form})$
  - Site Failure distribution during archival: $U(1/\phi_{site})$
  - Disk Failure Detection distribution: instantaneous
  - Format Failure Detection distribution: instantaneous
  - Site Failure Detection distribution: instantaneous
  - Disk Repair distribution: instantaneous
  - Format Repair distribution: instantaneous
  - Site Repair distribution: instantaneous

- Document creation: num_{doc} documents at startup, then no documents are created.
- Document access rate: irrelevant as failure distribution during access is the same as during archival.
- Access duration rate: irrelevant as failure distribution during access is the same as during archival.
- Document selection: uniform over the num_{doc} documents.

• **Policies**

- Document Creation policy: for each document, four materializations are created, 2 in each site. In each site, each materialization is created in a different disk and in a different format.
- Document to Materialization: read from any materialization.
- Failure detection algorithm: complete scan of all disk, formats, and sites every τ_{sto}, τ_{form}, and τ_{site} days, respectively.
- Damage Repair algorithm: discard bad component and replace with new component taking δ_{sto}, δ_{form}, and δ_{site} days, for disks, formats, and sites respectively.
- Failure prevention algorithm: none

• **ArchSim Parameters**

- Stop Condition: when losing the first document.
- Simulation time unit: days.

Appendix B

List of Styles, Substyles, and Tones

B.1 Styles

Rock	Rap	Electronica	Comedy
Jazz	Latin	Country	Easy
World	Blues	Gospel	Vocal
Newage	Class	Soundtrack	Reggae
Folk	Bluegrass	Avntg	Misc
Cajun	Child	Celtic	Holid
Spokn	Gay		

B.2 SubStyles

Alternative Folk	Alternative Pop/Rock	Pop/Rock
Hard Rock	Post-Grunge	Hardcore Rap
Album Rock	Heavy Metal	Soft Rock
Club/Dance	Urban	Gangsta Rap
Singer/Songwriter	East Coast Rap	Psychedelic
Rock and Roll	West Coast Rap	Adult Contemporary
Song Parody	Television Music	Rap-Metal
Indie Rock	Roots Reggae	Punk-Pop
New Wave	Prog-Rock/Art Rock	Sketch Comedy
Adult Pop/Rock	House	Punk Revival
R and B	Electronica	Dance-Pop
Alternative Metal	G-Funk	Teen Pop
Pop	Novelty	Southern Rap
Third Wave Ska Revival	Ska-Punk	Folk-Rock
Arena Rock	Latin Continuum	Trance
Musical Comedy	Rap-Rock	Swing
College Rock	Caribbean	Grunge
Techno	Quiet Storm	Latin Pop
Contemp. Country	Ragga	Post-Rock
Post-Punk	Ska	Rocksteady
Country-Pop	Pop-Metal	Golden Age
Dirty South	Euro-Dance	Jam Bands
Progressive Bluegrass	Experimental Techno	Rock
Pop-Rap	Trad Rock	Dancehall
Political Reggae	Comedy Rock	Hard Bop
Blues-Rock	Traditional Pop	Smooth Soul
Pop-Soul	Euro-Pop	Punk
Glam Rock	Experimental Ro	Alternative Rap

cont'd.

Dark Ambient	Hardcore Punk	Indie Pop
Funk	Vocal Pop	Alternative Dance
Trip-Hop	Traditional Country	British Blues
Synth Pop	Neo-Traditionalist Country	Standup Comedy
Soul	New Jack Swing	Proto-Punk
Modern Electric Blues	New Age	Neo-Psychedelia
Jazz-Rock	Tropical	Ethnic Fusion
Orchestral Pop	IDM	Illbient
Jamaica	Surf	Industrial Metal
Texas Blues	Southern Soul	Traditional Bluegrass
Classical	Crossover Jazz	Vocal Jazz
Film Music	Movie Themes	British Metal
Britpop	Motown	Show Tunes
CCM	Mood Music	Old-Timey
Country-Rock	New Traditionalist	Contemporary R and B
Emo	Funk Metal	Instrumental Pop
Hair Metal	Progressive Big Band	Prank Calls
Underground Rap	Urban Folk	Power Pop
Progressive Trance	Uptown Soul	Christian Punk
Standards	Contemporary Reggae	American Underground
Skatepunk	Progressive Country	Space Rock
British Invasion	Mexico	Brill Building Pop
Sunshine Pop	Mixed Media	Ambient Techno
Honky Tonk	Teen Idol	Space
Country Comedy	Shock Jock	Disco
British Psychedelia	Rave	Tejano
Funky Breaks	Blue-Eyed Soul	Blue Humor
Dirty Rap	Modern Electric Texas Blues	Nashville Sound
Worldbeat	”Jungle/DrumN Bass”	Big Band
Doo Wop	Political Rap	Techno-Tribal

cont’d.

Rock en Espanol	Slowcore	Country-Folk
Baroque Pop	Deep Soul	Progressive House
Soul-Jazz	Soundtracks	Black Gospel
Psychedelic Pop	Anti-Folk	Experimental
Electric Texas Blues	Cowboy	Fusion
Lo-Fi	Experimental Big Band	Garage Rock
Morning Radio	Observational Humor	British Trad Rock
Acid House	Outlaw Country	Creative Orchestra
Rockabilly	Big Beat	Contemporary Bluegrass
Bubblegum	Jangle Pop	Nostalgia
Goth Rock	Southern Rock	Impressionist
West Coast Blues	Avant-Garde Jazz	Progressive Electronic
Bop	Political Folk	Americana
Spain	Blaxploitation	AM Pop
Traditional Folk	Country	Ambient
New Orleans Jazz	New York Punk	Dream Pop
Adult Alternative	Philly Soul	Jazz-Pop
Tin Pan Alley Pop	Contemporary Blues	Ambient Pop
Bluegrass	Junkanoo	Merengue
French Rock	Classic Jazz	Chicago Soul
Nueva Trova	Nueva Cancion	Death Metal/Black Metal
Computer Music	Girl Group	Mod
Cuba	Latin Rap	Heartland Rock
Soundtrack	Bolero	Slide Guitar Blues
Vocal	Northern Soul	Jazz-Rap
Post-Bop	Electric Chicago Blues	Contemporary Instrumental
Boogie Rock	Avant-Garde	Merenhouse
Pop Underground	Roots Rock	20th Cent. Classical
United States of America	Party Rap	Rap
Retro-Soul	New Orleans R and B	Ballads

cont'd

India	Raga	Jazz-Funk
British Heavy Metal	French Pop	Progressive Folk
Electronic	British Punk	Smooth Jazz
Aussie Rock	Space Age Pop	Hip-Hop
Country Gospel	Italy	Western Swing Revival
L.A. Punk	Prewar Country Blues	British Folk
Jump Blues	Thrash	Noise-Rock
Acid Jazz	Salsa	Indian Classical
Speed Metal	Alternative Country	Prewar Blues
Folk-Pop	Trinidad	Dominican Republic
Tribal-House	Mod Revival	British Folk-Rock
Celtic Rock	Turntablism	New Romantic
Contemporary Folk	Christian Rock	Noise Pop
Surf Revival	Inspirational	Modern Electric Chicago Blues
Bachata-Merengue	World Fusion	Dixieland
Traditional Gospel	France	Africa
Tech-House	Delta Blues	Solo Instrumental
Chicago Blues	Rai	Jesus Rock
Europe	Instrumental Rock	Spy Music
Chamber Music	American Popular Song	Blues Revival
Musical Theater	Contemporary Jazz	Sophisti-Pop
Asia	TV Soundtrack	Jazz
Sweet Bands	Jazz-House	Sahel
Nature	Polka	Shoegazing
Ragtime	Zouk	Straight-Edge
Latin Jazz	Comedy	Freestyle
Hong Kong Pop	Dub	British Garage
Senegal	Gabba	Gospel
Marches	Cool	Cast Recordings
Brazilian Jazz	Electro	Traditional Scottish

cont'd

Puerto Rico	DJ	Goth Metal
Blues	Oi!	Folk-Blues
Dance Bands	New Orleans Blues	Euro-Rock
Bhangra	Ivory Coast	Merseybeat
Hi-NRG	Neo-Classical	Indian Diaspora
Free Jazz	Modern Acoustic Blues	Comedy Rap
Pub Rock	Brazil	Cabaret
Electric Country Blues	Piedmont Blues	Free Improvisation
Scandinavian Metal	Television Soundtracks	Hungary
Christian Metal	Jewish Music	Riot Grrrl
British Rap	Italian Pop	Hungarian Folk
Acappella	New Zealand Rock	Folk Revival
Celtic	Folk	Traditional
Ecuador	Middle East	Bakersfield Sound
Hardcore Techno	Praise and Worship	Celebrity
Goa Trance	No Wave	Urban Blues
Bass Music	Israel	Childrens
Greece	Calypso	Contemporary Singer
Hawaii	Yiddish	Cajun
Algeria	Ethiopia	Jewish Folk
Industrial Dance	Chamber Pop	Thailand
Close Harmony	Conjunto	East Coast Blues
Instrumental Country	Soca	Choral
Prewar Gospel Blues	Classic Female Blues	Lounge
Afro-Cuban Jazz	Singalong	Harmonica Blues
Christian Rap	Power Metal	Canada
Gypsy	Mali	Electric Blues
Read-Along Stories	Kora	Contemporary Gospel

cont'd.

Acoustic Texas Blues	Chamber Jazz	Frat Rock
Afro-Cuban	Irish Folk	Chanukah
Math Rock	Indian Pop	Guitar Virtuoso
Neo-Traditional Folk	Latin Rock	Soul-Blues
Twee Pop	Afro-Pop	England
Neo-Bop	Paisley Underground	Christmas
Acid Rock	Traditional Celtic	Experimental Ambient
Electric Harmonica Blues	C-86	Beach
Norteno	Musicals	Scotland
Social Commentary	Ranchera	Ambient Breakbeat
Progressive Metal	Poetry	Exotica
Modern Creative	Swedish Pop/Rock	Original Score
Industrial	Urban Cowboy	Alternative CCM
Alternative Country-Rock	American Punk	Reggae-Pop
Old School Rap	Sea Shanties	Latin Dance
Beat Poetry	Corrido	Folksongs
Country Boogie	Canterbury Scene	Noise
Piano Blues	Minimalism	Obscuro
Satire	Detroit Rock	Happy Hardcore

B.3 Tones

French Antilles	Brown-Eyed Soul	Skiffle	Ireland
Theatrical	Stylish	Playful	Energetic
PartyCelebratory	Confident	Earnest	Rowdy
Aggressive	Witty	Raucous	Laid-Back/Mellow
Rousing	Quirky	Reflective	Sophisticated
Passionate	Humorous	Fun	Sentimental
Brash	Poignant	Thuggish	Exuberant
Freewheeling	RefinedMannered	Irreverent	Organic
Boisterous	Cheerful	Trippy	Visceral
Menacing	Hostile	Wistful	Melancholy
Wry	Volatile	CynicalSarcastic	Intense
Rollicking	Soothing	Whimsical	Ominous
Cathartic	Summery	Sexy	Romantic
Carefree	Detached	Angst-Ridden	Provocative
Earthy	Brooding	Outrageous	Bittersweet
Reckless	Sleazy	Angry	Harsh
Sweet	Intimate	Manic	Sexual
TenseAnxious	Fiery	Gentle	Snide
Elegant	Ironic	Somber	Druggy
Happy	Hypnotic	Nihilistic	Eerie
Cerebral	Nocturnal	Restrained	Joyous
Gloomy	Complex	Acerbic	Yearning
Plaintive	Bleak	Autumnal	Wintry
Innocent	Searching	Spiritual	CalmPeaceful
Ethereal	Campy	Messy	Paranoid
Malevolent	Bitter	Lush	Precious
Sad	Reserved	Ambitious	Gleeful
Clinical	Self-Conscious	Dramatic	Smooth
Reverent	Urgent	Naive	Hedonistic

cont'd.

AmiableGood-Natured	Rebellious	Silly	Sensual
Rambunctious	Literate	Confrontational	Street-Smart
Enigmatic	Springlike	Gritty	Sparse
Indulgent	Delicate	Elaborate	Weary