International Journal on Digital Libraries manuscript No.
(will be inserted by the editor)

InfoMonitor: Unobtrusively archiving a World Wide Web server*

Brian F. Cooper, Hector Garcia-Molina

Department of Computer Science
Stanford University
e-mail: {cooperb,hector}@DB.Stanford.EDU

Received: date / Revised version: date

Abstract. It is important to provide long-term preservation of digital data even when that data is stored in an unreliable system, such as a filesystem, a legacy database, or even the World Wide Web. In this paper we focus on the problem of archiving the contents of a web site without disrupting users who maintain the site. We propose an archival storage system, the InfoMonitor, in which a reliable archive is integrated with an unmodified existing store. Implementing such a system presents various challenges related to the mismatch of features between the components, such as differences in naming and data manipulation operations. We examine each of these issues as well as solutions for the conflicts that arise. We also discuss our experience using the InfoMonitor to archive the Stanford Database Group's web site.

Key words: archiving – preservation – web pages – unobtrusive data collection

1 Introduction

Important and irreplaceable data is constantly being lost. For example, some data from the 1960 U.S. Census has disappeared, and files detailing land usage patterns and the location of natural resources in New York State are irretrievable now [11]. As more and more data is placed on the unreliable World Wide Web (technical papers, virtual museums, product catalogs), the losses of valuable information are sure to increase. The fundamental problem is that important data has been placed and will continue to be placed on storage systems (web servers, email servers, file systems, legacy database systems) that were not designed with data archiving as the primary goal. The designers of these *legacy* data stores focused on other design goals, such as efficiency, ease-of use, or reduced cost, usually to the detriment of the long term reliability of the system.

Our goal is to build an information archiving system that combines the best of both worlds, legacy and archival. This combined system should allow data to be accessed precisely as it is accessed today in the unreliable legacy system, so that all applications continue to run, and users are not aware of the reliability features unless they need them. At the same time, the combined system should preserve the information selected by a site administrator, storing data for historical purposes and also making it possible to restore data into the unreliable system when necessary.

Let us see how we can build such a combined system. Figure 1 illustrates four approaches that might be considered. The first option (Fig. 1a) is to redesign the legacy system to have both the traits of the unreliable data store and a reliable archive. This of course is an expensive solution if we must build a system from scratch. Even if there is an existing code base, we have to get access to the code and permission to modify it. In either case, the resulting hybrid system may not even be backwards compatible in the end.

The remaining three solutions avoid these problems by integrating the existing legacy system with a component built for archival storage. In the second solution, Fig. 1b, an archival repository module intercepts calls to the legacy system, and performs necessary actions to preserve information. For example, if a file is updated, the previous version may be automatically saved (perhaps in a different underlying legacy system for added reliability). In order for users to access the system, their applications would have to be relinked to access the reliable archive module. Even if users have the source code for all of their applications, the new interface may require significant application rewriting, and they may not

* This material is based upon work supported by the National Science Foundation under Award 9811992.

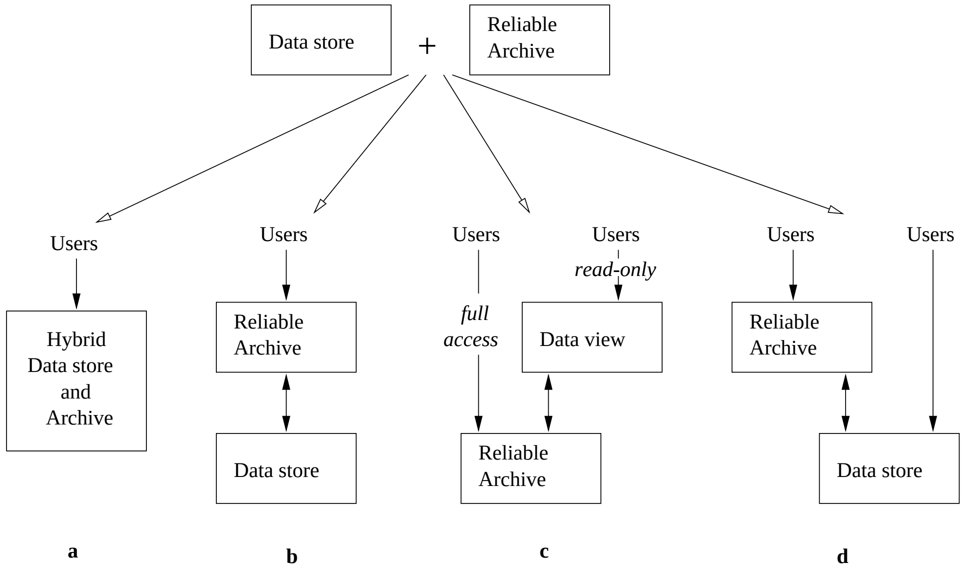

Fig. 1. Options for combining an existing data store with an archival component.

be willing to deal with the cost and effort to redeploy applications. In the third solution (Fig. 1c), the data is actually moved to a reliable archival system, and a read-only view is made available on the legacy system. This approach is used today by many web sites, which store their base data in a database, and generate HTML views for their web site. The main disadvantage is that only read access is provided through the original legacy system (the web site). Changes to the data must be directed to the new system, which once again involves changing any application that updates the data.

Finally, the fourth solution, Fig. 1d, is to take *existing* legacy and archival systems and combine them so that each still offers its native services. Users are free to access legacy data in traditional ways, while the archival system "tracks" changes in the legacy system and "translates" them into equivalent archiving operations in order to preserve the information. Although all four alternatives have strengths and may be the correct choice for certain domains, there are situations where the first three options are not available. If existing systems and applications should be retained unmodified, the alternative (d) offers flexibility for accomplishing archiving without disrupting usage. This is the design we examine here.

In order to implement solution (d), we must build a component that implements the "arrow" in Figure 1d between the data store and the archival system. We have designed and implemented the *InfoMonitor*, a module that archives data from a standard filesystem into the Stanford Archival Vault (SAV) [5], a reliable data archive. To illustrate the challenges and our solutions we will focus on the case study of a reliably archived web site. Many web sites still store pages on a standard filesystem rather than a DBMS or reliable archive. This allows users to modify these files using their favorite applications, including text editors and graphics software, while providing a convenient environment for server side programming, such as CGI scripts. However, filesystems are notoriously unreliable. Thus, it is necessary to archive the data served by the web site in a reliable database system. If the web site authoring and management software does not provide such archiving capabilities (and most packages do not; see Sect. 8), a reliable archive should be integrated with the filesystem itself, and that is the design we adopt here.

Here, we use a filesystem as an example legacy store. However, the InfoMonitor architecture is applicable to archiving data on any legacy store, although modifications to the implementation may be required in order to use the interface provided by the legacy store. For example, archiving externally controlled web sites or dynamic content requires a new component to retrieve the appropriate objects, e.g. via HTTP. However, the basic architecture of the InfoMonitor as a bridge between the web site and the archive is retained. We discuss extending the InfoMonitor for this purpose in Section 7.

In this paper we consider the various problems that arise when integrating an archiving system with a traditional data store, focusing on the situation where the data store cannot be expected to provide active assistance for archiving. Specifically, we make several contributions:

- A proposed architecture for adding archiving capabilities to an existing data store. This architecture is flexible and provides archiving transparently.
- Solutions for specific issues that arise when implementing this architecture, including seamless translation of data features and naming.

– An implemented system, called the *InfoMonitor*, for the archived web server situation, but which is applicable to other domains as well.
– Discussion of performance and scalability issues. The system performs well on a large-sized web site, and this performance scales linearly with the size of the archived data set.

Our system is similar in many ways to previous work in areas such as data backup and file versioning. Unlike these previous systems, we focus on constantly monitoring a passive filesystem, copying all updated files to a reliable, heterogenous store. We will discuss these differences in more detail in Sect. 2 after we present an overview of our system.

The paper is organized as follows. First, we examine the architecture of the InfoMonitor in Sect. 2 by examining the steps users go through to archive data. Next, we examine issues that arise when implementing the InfoMonitor. Section 3 illustrates the problem of detecting filesystem changes and representing those changes in the archive. Section 4 examines the problem of representing filenames in the archive, while Sect. 5 examines the situation where only a subset of the filesystem's data should be archived. In Sect. 6 we present our experience using the system to archive our own web site, including performance measurements. Sect. 7 discusses extending the InfoMonitor to archive external web sites and dynamic content. Finally, Sect. 8 examines related work and in Sect. 9 we outline our conclusions.

2 The InfoMonitor

This design of our system is shown in Fig. 2. Each of the three boxes in the figure represent a distinct component that can run on its own machine. The standard data store is an existing filesystem supporting a web server. (See Section 7 for the situation where the filesystem cannot be accessed directly or we must archive external resources.) If the InfoMonitor runs on a separate machine from the web server, it can access the server's files using a network filesystem. The InfoMonitor accesses the archive using the archive's native protocol. The InfoMonitor can work with a variety of archives, and the archive may store other data sets not managed by the InfoMonitor.

The InfoMonitor is started by a system administrator who specifies which portions of the filesystem should be archived. The specification gives a start point for a depth-first traversal of the filesystem, and a *filter set* that determines which files are archived (see Sect. 5). The start point for archiving a web collection is usually the root of the public_html hierarchy. In order to reach individual user's public_html directories, the InfoMonitor can utilize symbolic links from the central public_html directory (if they exist) or can start at the root of the filesystem and filter out non-WWW specific directories. The administrator also specifies the archive to use. After startup, the InfoMonitor runs unattended, and the administrator only intervenes to restore data after a failure.

Once the InfoMonitor is started, it scans the filesystem, transferring all of the files that pass its filter set into the archive. For each file, the InfoMonitor archives the content of the file as well as some metadata (e.g., the filename). Issues related to managing filename metadata are discussed in Sect. 4. Moreover, the InfoMonitor archives the filesystem's directory hierarchy as a hierarchy of objects in the archive. The InfoMonitor also keeps an in-memory "summary" of the files (including a content signature) to be used to detect new and modified files.

After startup, the InfoMonitor unobtrusively monitors file activity by continually scanning the filesystem. When a user creates, modifies or deletes a file, the InfoMonitor detects this event as a discrepancy between the filesystem state and its in-memory summary, and copies the affected file into the archive. Changes are represented in the archive using version chains. In particular, deleted files are a treated as a new, "empty" version. This process is described in more detail in Sect. 3, where we also discuss the challenge of efficiently scanning the filesystem.

Because each created object is also timestamped, the archived files can be viewed in two ways: historically as files change over time, or as an instantaneous filesystem snapshot. The InfoMonitor provides a GUI for both of these perspectives. The *files view* shows the historical chain of modifications; a screenshot from our operational system is shown in Fig. 3. Files are displayed hierarchically, as they exist in the directory structure of the filesystem. Each folder icon represents a directory, while each "stacked document" icon represents a version chain. The user can expand a directory to view its contents, and can expand a version chain to see the individual versions. Each version is represented by a "single document" icon and the timestamp when that version was archived. In the figure, the file `cooperb.html` exists in three different versions. The user can select a version and click the "View" button to see the version's contents. The files view allows an administrator to navigate the history of the filesystem on a file by file basis.

A second view, the *snapshot view*, displays the entire state of the filesystem at a specified point in time. An example screenshot is shown in Fig. 4. The snapshot is organized hierarchically using the directory structure. The time and date at the bottom of the window is the instant at which the snapshot was valid. Users can update the snapshot to the current filesystem state using the "Reset date" button, or can specify a point in the past in the "Date" box and click the "Apply date" button. Users can select a file and click the "View" button to examine the file contents.

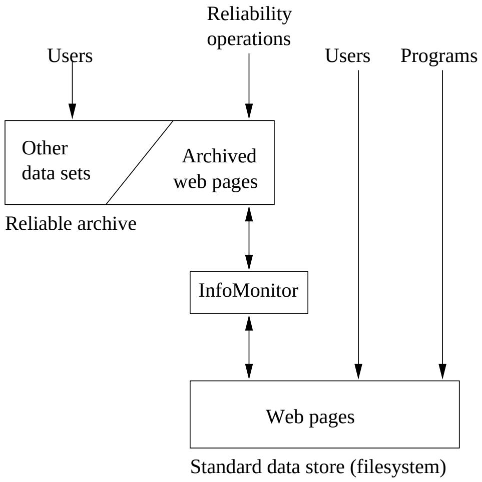

Fig. 2. The architecture of our implemented InfoMonitor system.

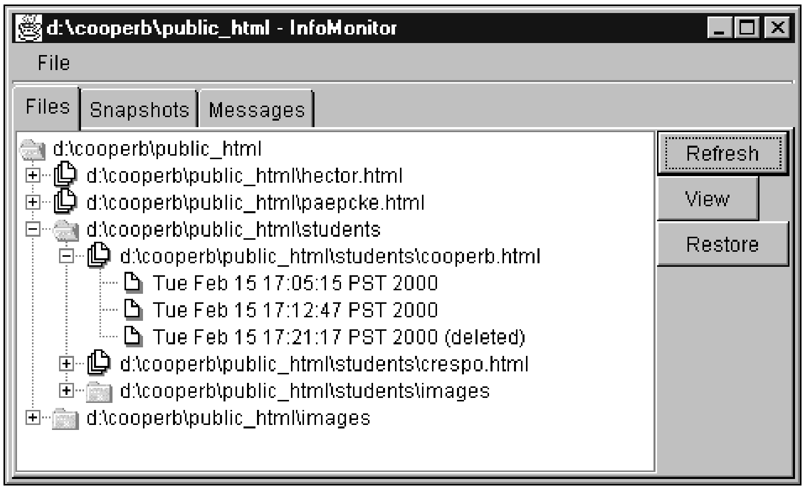

Fig. 3. The *files view*.

Note that the archiving process is both transparent and comprehensive. It is transparent because users continue to access and modify files using their favorite applications, and do not expend extra effort to achieve archiving. It is comprehensive because it covers objects not available through the HTTP interface to the web site, such as CGIs and scripts that generate dynamic web pages. (See Section 7.)

Human intervention is only necessary in the event of a filesystem failure. Both the *files view* and the *snapshot view* provide facilities for quickly restoring data. In each case, the restored data is treated as a "new version," allowing the InfoMonitor to maintain a simple linear version chain. The "Restore" button in Fig. 3 can be used to restore previous versions of individual files. In contrast, the "Restore" button in Fig. 4 restores an entire snapshot. Restoring a snapshot is tricky because it is possible to overwrite portions of the filesystem not covered by the snapshot, especially if the filter has changed over time. This issue is discussed in Sect. 5. Failures of the archive are handled by its own internal mechanisms without intervention from the InfoMonitor.

It may be necessary to restart the InfoMonitor because of crashes or upgrades. On startup, the InfoMonitor writes a "bookmark" into the archive referencing the archived directory hierarchy. When the InfoMonitor is restarted, it reads the bookmark, traverses the archived hierarchy, and reconstructs its internal file summaries. After this recovery process, it continues scanning the filesystem looking for file modifications. This recovery process is efficient because the InfoMonitor only needs to examine its own archived objects, not the entire archive.

2.1 The Stanford Archival Vault (SAV)

In our prototype, the archiving system is the Stanford Archival Vault (SAV). In this section, we give a brief overview of the SAV. For more details about the design and implementation of SAV, see [5].

The SAV system has the following characteristics:

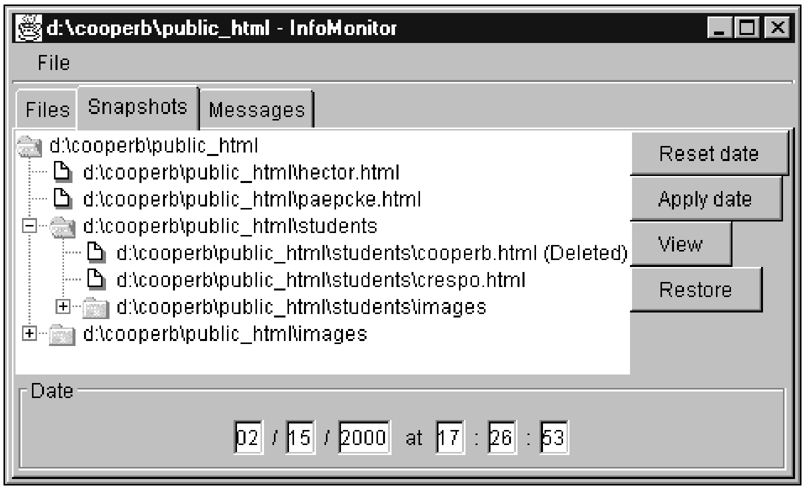

Fig. 4. The *snapshot view.*

- Objects are constructed as a list of tagged fields, containing either data or metadata.
- A signature function is used to create object names. Specifically, a 64 bit CRC is calculated for every object and used as the name. The signature provides several functions, including
  - A location service: signatures are used to locate objects in the store. The SAV implements a mechanism that maps from signatures to physical storage locations, so objects can be retrieved via signature.
  - Error detection: signatures are cached and periodically checked against the object; differences indicate object corruption.
  - Efficient object comparison: objects are identical if their signatures are identical.

  Although there is the possibility of signature collisions between different objects, we can minimize this possibility; see below.
- Metadata fields in objects can also contain signatures, which are interpreted as references to other objects. This provides the capability to create graph structures of objects.
- The store provides an interface to create new objects and read existing objects. However, data, once written, cannot be deleted or overwritten. The goal is to prevent data loss due to accidental or intentional data erasure. Moreover, this policy eliminates ambiguities from distributed, replicated data management, such as the need to determine which object is the "correct" object if different sites can modify it simultaneously.
- Reliability is provided by automatic replication and periodic comparison of data between several SAV sites. Corrupted or missing objects at one site are automatically replaced with pristine copies obtained from other sites.

We can minimize the probability of signature collisions between different object by increasing the number of bits in the signature. In [10] it is reported that for a collection with n objects and a signature size of b bits, the probability that no signatures collide is:

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

This expression gives us, then, the probability that the corruption of an archived object would not be detected, or that two different objects could be confused for the same object by the archival system. In our implementation, we chose a signature size of 64 bits. For the experiments reported in Section 6, where approximately 30,000 objects were archived, the chances of a signature collision are less than 3×10^{-11}. For larger collections, larger signatures are required. For example, 64 bits is not appropriate for a collection with 10^9 objects, since there is a 0.03 chance of collision. However, a 96 bit signature for a collection of 10^9 object ensures that the probability of collision is less than 6×10^{-12}.

If the probability of collision must be zero, the SAV can add extra bits to the signature that a unique ID for the SAV site. Then each SAV site must only ensure that signatures are unique locally, and can add extra bits (e.g. a sequence number) to the object if necessary when computing the CRC to ensure uniqueness. See [10] for more details.

2.2 Differences with existing approaches

As mentioned in the introduction, there has been a lot of work on tracking changes and failure recovery. For example, a data warehouse monitors data sources and copies changes to a centralized store [1,20,30]. In our context, the filesystem is the data source, the reliable archive is the warehouse, and the InfoMonitor is the data extractor. However, unlike data warehousing, the InfoMonitor focuses on the ability to restore data to the original source. Thus the InfoMonitor must save information necessary to map archive objects back into the corresponding source objects. Furthermore, the InfoMonitor is designed to deal with very large numbers of relatively small files (e.g., web pages) as opposed to a few very large files (e.g., tables containing sales records). As a result, the techniques for extracting changes are different than the ones used in say [21]. Finally, the focus of a data warehouse is to provide processing, not archiving. While it is possible for a warehouse to provide data histories, the primary focus of the InfoMonitor is to store complete historical snapshots of the archived information.

Our work is also related to file backup systems [3]. While file backup systems are in widespread use, there have not been many academic studies of backup systems. In addition, backup systems tend use the same data model as the filesystem (e.g., files are named, unstructured sequences of bytes). In our system, the archive can use any data model, and the InfoMonitor performs the translation from file objects to archive objects. This supports extensibility to other data sources, besides filesystems (See Section 7.2). Moreover, we focus on constantly scanning a filesystem to record as many changes as possible, while a backup system may only run periodically (e.g. once a day or once a week). The InfoMonitor GUI also offers fine grain control for browsing and restoring files.

The InfoMonitor borrows concepts from previous versioning technologies [13,29]. For instance, versioning systems like RCS provide a way to maintain a history of changes to files. However, there are three important differences with "traditional" versioning systems. One, the InfoMonitor tracks an *autonomous* filesystem, and stores the versions on a separate system. In a traditional system, the versioning mechanism is notified whenever an object is changed. In contrast, the InfoMonitor must detect changes itself and must minimize the number of missed changes. Two, the filesystem and the archive use different object representations. For example, if the name of a file is changed, this may generate a different action at the archive, such as "new object" or "update metadata." Three, the InfoMonitor can use a *simple* version mechanism targeted to file restore only. That is, the InfoMonitor does not need to consider alternatives but can store versions as a linear chain. (There is always an unambiguous "current version" instead of multiple alternate current versions created by branches in the version tree.)

Other related work is discussed in Sect. 8.

3 Data manipulation

The archive should record information about operations performed on the filesystem, even though those operations are not native to the SAV. Specifically, the following types of information should be archived:

- *content*: Data contained in the contents of files.
- *structure*: Information about relationships between files.
- *events*: Information about the sequence of modifications to a file.

An archive must obviously retain file content, but structure between files is equally important. For example, when archiving a web site it is important to archive the hyperlink structure between HTML pages, so that if the web site must be reconstructed all of the links still work. This requires the InfoMonitor to both detect structures and record them in the archive. Unfortunately, in a filesystem the only structure that is explicitly represented is the hierarchical organization of files in directories. Other structures, such as hyperlinks, are represented in an application specific manner, and the InfoMonitor must take steps to interpret and preserve this structure information. For now, we take advantage of the fact that hyperlinks between files utilize the file naming convention. Then, effectively preserving file names allows us to reconstruct the hyperlink structure. If necessary, we can extend our system to parse the HTML pages, extract the links and maintain them explicitly.

Finally, the sequence of events should be recorded so that if necessary certain events can be undone. For example, if a user accidentally erases or overwrites important data, it should be possible to select that event to be undone, while preserving previous intentional modifications. The concept of a version chain (such as in versioning systems like RCS [29]) can be used to represent filesystem events as a sequence of resulting file versions. However, as mentioned earlier, the filesystem does not provide explicit notification when a new version has been created (e.g., there is no file "check-in"). Note that here we treat a file creation as the start of a new version chain, and file deletion as a modification which produces an empty version.

3.1 Detecting file modifications

The InfoMonitor must detect and report *filesystem events*: new files, modifications and deletions. The InfoMonitor constructs a summary of the filesystem when the monitor is first started, and then continually scans the filesystem, comparing the summary to the files. A discrepancy between the summary and the filesystem indicates a filesystem event, causing the InfoMonitor to migrate the affected data into the archive and update the summary. Because we expect that the filesystem will contain a large amount of data, the summary must be small and facilitate time-efficient data comparison, or events will be missed.

We have divided the scanning process into two components:

– *quick scan*: uses filesystem timestamps to indicate events
– *slow scan*: compares file contents using a signature algorithm

Both scans start at an initial location in the filesystem, and perform a depth first search of the directory structure. The selection of initial location and the nature of the search is further described in Sect. 5.

In order to perform the *quick scan*, the InfoMonitor maintains in its in-memory summary a list of file names along with the filesystem timestamp for each file. By relying on timestamps, the InfoMonitor is able to quickly scan the filesystem for events and archive affected data. The quick scan operates as follows:

– A file on the filesystem that is not in the summary is new. The filename and timestamp are added to the summary and the file is written to the archive.
– A file that is in the summary but does not exist on the filesystem has been deleted. The filename is marked as "deleted" in the summary and an "empty version" is archived.
– A file that is "deleted" in the summary but exists on the filesystem has been undeleted. The deletion mark is removed from the summary, and the undeleted version is archived.
– If a file is in the summary and on the filesystem, then the filesystem timestamp is compared to the timestamp stored in the summary. If the filesystem timestamp is newer than the summary timestamp, the file has been modified. The summary timestamp is updated, and the new version of the file contents is archived.

Unfortunately, timestamps on the filesystem cannot always be trusted. The primary reason for adding the reliable archive is that objects in the filesystem can become corrupted, including objects containing timestamps. There are also other factors which may reduce the reliability of timestamps, such as clock skew between hosts on the same network filesystem.

Therefore, the InfoMonitor also performs the *slow scan*. The slow scan operates using the same steps as the quick scan. However, instead of comparing timestamps, the slow scan compares the current contents on the filesystem to the last version that was archived. Differences indicate that the file has changed and that the new version should be archived.

We examined the possibility of reading each file from the reliable archive and comparing it to the filesystem file to detect changes. However, it is prohibitively expensive to read all of the files out of the archive, especially since the SAV resides on a different machine than the InfoMonitor, and the information must be transferred via the network. Moreover, we cannot cache the contents of every archived file in RAM to detect changes, since the main memory of the machine is not large enough.

In order to achieve reasonable performance, we use the following *signature-based* process:

1. The 64 bit CRC of every new or modified file detected by the quick scan is calculated and stored in the summary (together with the filename and timestamp).
2. The slow scan calculates the CRC for every file on the filesystem, and compares it to the CRC in the summary.
   – If the signatures are different, the file has changed and the new version must be archived. The new signature replaces the old signature in the summary.

Note that if the signatures are different but the timestamp has not changed, this is a hint that the file has been corrupted, since every "normal" filesystem operation should update the timestamp. The InfoMonitor prints a message to the user that the file may have been corrupted, and the user can take appropriate action.

The signature-based scheme has two potential disadvantages. First, the signature-based scheme may miss file modifications if the signature of the modified file is the same as the signature of the original file. The nature of the CRC algorithm allows us to make the probability of such false negatives arbitrarily small by using enough signature bits. Second, signatures stored in memory may become corrupted due to memory errors or be lost if the InfoMonitor host crashes. The InfoMonitor can be restarted in these cases and will recalculate all of the signatures based on the data reliably stored in the archive. Alternately, the content signatures could also be archived and retrieved periodically at less cost than retrieving the entire data set.

The InfoMonitor runs the quick scan much more frequently than the slow scan (which is much more time consuming). The quick scan will detect most of the filesystem events. A few modified files will be missed due to timestamp problems but these files will be caught by the slow scan. A few events will still be missed because the quick scan is not instantaneous, and a file may change several times between repeated examinations. However, it is unlikely that many vital changes to a file will be made in the interval between file examinations as long as the interval is small.

4 Naming

The archive must retain file names as reliably as the file contents. Ideally, the InfoMonitor would store a *naming history* for each archived file. There are several challenges that arise when dealing with naming:

– It is difficult to determine when a file's name has changed.
– It must be possible to locate archived objects using their filename.

– Files can have aliases.

We examine each of these issues in this section.

4.1 Detecting name changes

The InfoMonitor must update the archive after a name change. As with the data modifications discussed in Sect. 3.1, the InfoMonitor cannot expect explicit notification of a name change. Unlike file modifications, however, the quick and slow scans are unable to detect name changes. The problem is that it is difficult or even impossible to distinguish between the case where a file's name changes and the case where one file is deleted and another, with similar content but a different name, is created.

One solution would be to utilize some sort of *arbitrator* that can determine whether two files are the same. In Unix, files are identified by an inode number that does not change across simple name changes. Two files with the same inode number are the same, even if the filename changes. This solution is limited in applicability to Unix filesystems; other filesystems may not have inodes.

The approach we take in the InfoMonitor is to view a file renaming as a deletion of one file followed by the creation of a different file. Thus, the InfoMonitor updates the archive after a name change but does not keep an explicit naming history for individual files. The domain of archiving a web server suggests our approach is appropriate. From the perspective of a web server, objects with different names are in fact different objects, since a name change requires links to be updated to point to a new URL containing the new filename. Therefore it is not necessary to resort to an arbitrator to determine if objects with different names are the same object; we simply assume that they are not.

4.2 Locating archived files by filename

The SAV uses its own naming scheme for objects (specifically, content-based signatures), and does not provide a direct mechanism for finding a file based on the filesystem name. However, users may need to find archive objects based on the filename. For example, to restore a file to the filesystem, the user should be able to specify the filename, not the SAV signature. The obvious solution is to build a mapping from filename to SAV signature by loading every SAV object and examining its filename field. We can avoid this expensive process by reusing the in-memory summary maintained by the InfoMonitor. By adding an "archive signature" field to the summary, we can look up a file name, and find the associated SAV signature. Then, we can retrieve the object from the SAV using the archive's service for retrieving objects via signature.

4.3 Filesystem aliasing

Filesystems provide the ability for the same file to have different names. Usually, the InfoMonitor can treat two different names as two different files (for the same reason it ignores renaming; see Sect. 4.1). However, symbolic links can create cycles in the filesystem graph. Such cycles can create an infinite number of aliases for the same file. For example, if directory `/dir` contains a symbolic link named `subdir` which points to `/dir`, then every object named `file` in `/dir` is simultaneously named `/dir/file`, `/dir/subdir/file`, `/dir/subdir/subdir/-file`, and so on. Because we regard each name as a different file, there would be an infinite numer of files to be archived. We cannot ignore symbolic links because they may form a vital part of the filesystem scan (e.g., a symbolic link from the central public_html directory to a user's personal public_html directory may be the only way to reach that user's files from the InfoMonitor's start point).

We resolved this issue by reducing symbolic links to check for previously scanned objects. For each file and directory, the filesystem is queried for its *canonical name*: the name constructed from the directory hierarchy DAG that results from removing all symbolic links. If `dir` is an actual directory and not a symbolic link, then the canonical name of `file` is `/dir/file`. When `/dir/subdir` is encountered, it is reduced to its canonical name, `/dir`, and ignored since the InfoMonitor has already scanned `/dir`.

Moreover, we can archive the symbolic links themselves. On a Unix filesystem, a symbolic link is a file that contains a reference to anothre file. We archive the symbolic link file, without following the reference. This allows the InfoMonitor to record the symbolic link structure without having to follow all symbolic links (and possibly enter an infinite cycle.)

5 Determining what to archive

It is possible to archive the entire contents of the filesystem, but it is often desirable to archive only a subset. In a web server, the same networked filesystem that contains the web data also contains applications, the operating system, private user data, etc., none of which should be archived. The administrator should be able to configure the InfoMonitor to select a particular set of files to archive. This *logical set* should contain semantically related items (e.g., all web pages and associated data).

One way to construct this set would be to maintain a list of all files to be archived. This scheme would require frequent human intervention to add newly created files to the list, a requirement we want to avoid. Another option is to specify a directory, and have the InfoMonitor archive every file in this directory and its subdirectories. This option may not be appropriate if there are files in that directory tree that should not be archived, or if files are scattered throughout the filesystem in multiple directories.

Another option (which we do not use) is to utilize the hyperlink structure contained in a set of web pages. The logical set could be defined as all web pages reachable via hyperlinks from a starting web page. Any time a user modifies a page in the existing logical set to link to a new page, that page would be implicitly added to the logical set. It may be necessary to stipulate that the hyperlink traversal does not cross filesystem boundaries, or that it extends no more than n links from the start page. This may be the correct approach if the InfoMonitor archived pages retrieved via HTTP instead of filesystem files.

We use simple querying to allow an administrator to define what kinds of files should be archived without having to list all of the files explicitly. The InfoMonitor uses simple querying in the form of a *filter set*: a sequence of regular expressions, each of which filter out certain files based on the absolute path and filename of the file. (This is similar to filter sets in archiving programs such as *tar*.) This process gives an administrator flexibility in designing a logical set. For example, the filter language makes it possible to accept or reject all names except a specified few.

This filter set is applied by the InfoMonitor during both the quick and slow scans to both select individual files in the logical set and to prune the filesystem search tree. When the InfoMonitor finds a directory, it applies the path name of the directory against the filter set and recurses into the directory if the name passes. When the InfoMonitor finds a file, it applies the path and filename against the filter set and examines the file if the path and filename pass.

The administrator does not need to change the filter set once it is specified. New files that are created will be archived if they pass the filter set, allowing the logical set to grow implicitly. However, we do allow the administrator to change the contents of the filter set if necessary. Recall from Sect. 2 that a "snapshot" of the entire logical set as of a point in the past can be restored; the set of restored items is defined by what filter set was used when the snapshot was valid. Therefore, it is important to store a timestamped version chain of the filter sets. A filter set is just a file that is automatically added to every logical set and archived like other files.

We use the following criteria to determine how a snapshot is restored, based on the filter set that was active at the time of the snapshot:

- A file present in the snapshot must have passed the filter set, and is restored to the version it had when the snapshot was valid. This includes files that no longer exist on the filesystem.
- A file that was "deleted" when the snapshot was valid is only deleted from the filesystem if that file would have passed the filter set for the snapshot. A change to the filter set can cause the file to be removed from the logical set, and the InfoMonitor would record this as a "deletion" of the file. Files removed from the InfoMonitor's jurisdiction should not be affected by the restore.
- Files present in the filesystem but not mentioned in the snapshot are not affected.

Thus, the version chain of filter sets is critical to the snapshot restore process.

6 Performance measurements

We have measured the performance of our implemented system. Our goal is not to produce the most efficient implementation or a production-level system. Instead, we wish to demonstrate a proof-of-concept and to explore performance. Nonetheless, we have attempted to construct a system that performs well. We have focused especially on performing the quick scan as rapidly as possible to reduce the number of missed events (see Sect. 3.1).

The InfoMonitor system, including the user interface, consists of 3,400 lines of Java 2 code. The monitor communicates with the SAV, also written in Java, using Visibroker CORBA. Table 1 describes the hardware we used to collect performance data. The InfoMonitor and the SAV ran on different machines connected via 10 Mbps ethernet. This models the situation where the SAV is running on an archiving host and the InfoMonitor serves as a remote client. The InfoMonitor machine and the HTTP server machine share the same DFS filesystem.

We used the InfoMonitor to archive the files from the Stanford Database Group's web site. The web data consists of over 24,000 files (1.6 gigabytes). In addition to text HTML files, this data set contains images, programs, compressed files, word processor documents, raw data files, and a variety of other file types. The size and diversity of this filesystem allowed us to test the system for both scalability and robustness for different kinds of data. While the InfoMonitor was running, users were actively accessing and modifying files. The measurements for the phases of the InfoMonitor's operation are shown in Table 2. (See Sect. 3.1 for descriptions of these tasks.)

We believe these measurements are appropriate for an effective demonstration of the system. First, the files represent a wide variety of data produced by many different users. Moreover, the data set represents an average sized or larger web site. A study by Inktomi reports that on average, a website contains 200 HTML pages [7]. Our group's web site contains over five times as many pages, as well as many more non-HTML files (representing 23 graphics, scripts, downloadable documents, etc. per HTML page). Therefore, running the InfoMonitor on our site reveals the performance of the system for a typical web site.

Much of the time required to perform the archiving is unavoidable. To illustrate this point, we list the breakdown of times for the initial load phase in Table 3. An

	InfoMonitor	**SAV**	**Web Server**
System	Sun Ultra 2	IBM Intellistation	Sun Ultra 2
Processor	Dual 450 mhz UltraSPARC	450 mhz Pentium II	Dual 200 mhz SPARC
Memory	256 MB RAM, 1800 MB swap	256 MB RAM, 512 MB swap	256 MB RAM, 1230 MB swap

Table 1. The hardware specifications of the machines used to collect performance data.

Logical Data Set	Period	24 contiguous weekday hours
	Number of files	24,885
	Total size	1.6 GB
Initial Load	Duration	2 hours, 8 minutes
Quick Scans	Average duration	5 minutes, 20 seconds
	New files detected	5,020 files (155 MB)
	Modified files detected	14
	Deleted files detected	14,462
	Undeleted files detected	9,452
Slow Scans	Average duration	55 minutes, 31 seconds

Table 2. The results of running the InfoMonitor on the entire web site.

Task	**Time (seconds)**	**Percent**
Read files from disk	722	9.3 %
Build summary	199	2.6 %
Buffer copy	2124	27.4 %
Network transfer to SAV	3340	43.1 %
Store in SAV	1373	17.7 %

Table 3. The breakdown of the initial load time.

"ideally performing" archiving system would still have to read files, transfer them to the archive, and save them in the archive's storage. According to Table 3, this unavoidable overhead accounts for over 70 percent of the initial load time. Of the remaining 30 percent, 27.4 percent is for copying file data into messages sent to the SAV; this "buffer copy" could be avoided with a more streamlined implementation. Building the in-memory filesystem summary only adds 3.5 percent overhead to the "ideal" archiving system, indicating that the InfoMonitor (other than the buffer copy) is very efficient.

To explore scalability, we used the InfoMonitor to archive data sets ranging from one-third to almost twice the size of our web site. These data sets were produced either by taking a subset of the 24,000 files in the group's site, or by duplicating certain files[1]. Therefore, each of these data sets contain similar mixtures of file types. The measurements are shown in Fig. 5, where we have plotted the total time for each task against the size of the data set. The InfoMonitor's performance scales linearly with the size of the data archived.

Another interesting result shown in Fig. 5 is that the quick scan duration increases very slowly as the data set gets larger. This is desirable, since the quick scan is the task performed most frequently. The use of the quick scan as the primary event detection mechanism results in a system that scales well to larger data sets.

Although the performance of our system scales linearly to data sets on the order of a few gigabytes, it is reasonable to ask whether this linearity will continue to much larger data sets (hundreds of gigabytes or larger). We believe it will. First, each of the data sets shown in Fig. 5 is larger than the 256 megabytes of physical RAM, and two of the data sets are larger than the total swap space available (1,800 megabytes). Thus, working with data sets that do not fit in main memory (or even swap space) does not hinder linear scalability. The nature of the algorithms we use require individual files to be loaded and examined; since we do not join information between files we do not keep more than one file in memory at a time. Even if a file is large it can be examined in chunks. Simply examining more files should not introduce nonlinearity in performance.

The InfoMonitor keeps an in-memory summary of the files, and we can ask what will happen when the summary no longer fits in memory. Only 50 bytes of memory is required per file[2], and if we allocate 128 megabytes of main memory as the maximum summary size, the InfoMonitor can index in main memory 2.7 million files

[1] The InfoMonitor treated aliased files as a fixed number of different files, avoiding the infinite looping discussed in Sect. 4.3. Aliased files thus appeared as different objects, and the effect is that the data set contained more files.

[2] Each entry contains the filename, timestamp, content signature, a "file deleted" flag, the SAV object handle, and a pointer to the path name. The cost of shared storage of path names is amortized over an average of 20 files per directory. The sum of the fields, amortized cost of the path name, and hash table overhead is 50 bytes.

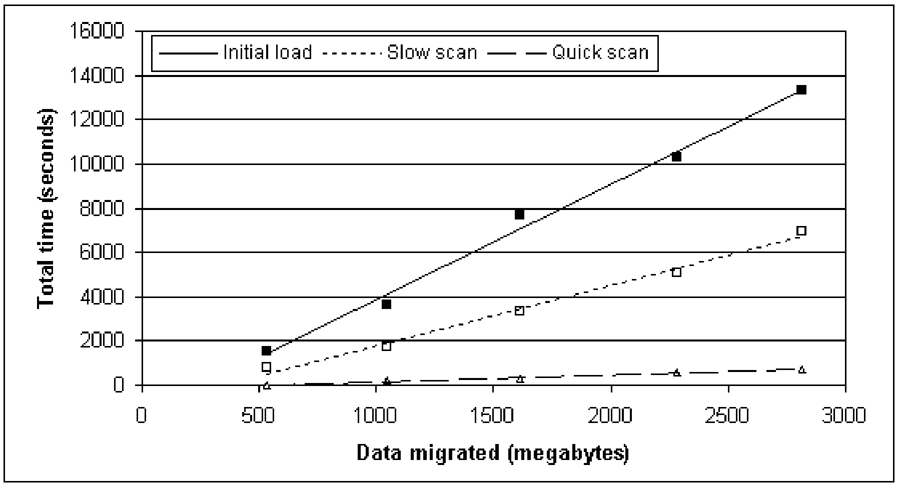

Fig. 5. The total time of the InfoMonitor's tasks as a function of the size of the data set.

(roughly 170 gigabytes). For larger data sets, it is either necessary to add more RAM (which is relatively inexpensive) or to store some of the summary on disk. The InfoMonitor can organize a disk index using the order in which files are examined (based on a depth first traversal of the filesystem), and can load the index sequentially into memory based on the files currently being scanned. As a result, even an index stored on disk incurs a fixed cost per file archived, and the performance of the InfoMonitor for data sets larger than 170 gigabytes (with 128 megabytes of RAM) should be linear in the amount of data archived.

There are a few more conclusions we can draw from our experiments. First, the duration of the quick scan means that some filesystem events are missed, and we would like to decrease the quick scan duration. One solution is to divide the data set into subsets, and start a different InfoMonitor process for each subset. The processes could operate in parallel, possibly on different machines with the same networked filesystem.

Second, it is apparent that the InfoMonitor is responsible for detecting events in a small fraction of the filesystem. Although the quick scans detected almost 29,000 filesystem events in a 24 hour interval, 86 percent of the events were localized in 10 percent of the directories, and 65 percent of the directories experienced no events at all. Perhaps the scans can be optimized to focus more precisely on the "busy" directories; for example:

- Increasing the frequency of scans of "busy" directories while performing fewer scans of directories whose contents never change.
- Running multiple instances of the InfoMonitor. Each instance can quickly scan a small portion of the filesystem, so we could dedicate an instance to each quickly changing portion of the filesystem and one instance to cover all of the less busy portions.
- Calculating a signature of the sequence of timestamps of files in a particular directory, and performing a file-by-file comparison only when the signature changes. This is similar to the signature snapshot comparison algorithm described in [9].

Finally, whenever a file is undeleted or modified (even slightly), the entire file is copied into the archive; 9,466 files were archived for this reason. The InfoMonitor could reduce space usage in the archive due to these new versions by creating a "diff" that represents only the changes to the file. Another option is to utilize a compression algorithm (such as the Lempel-Ziv coding [31] used in gzip) to reduce the space requirements of all files. Compression could be time consuming, and should not be done during the time-sensitive quick scan. Instead, the InfoMonitor could archive entire files, and then ask the archive to lazily compress all versions during periods of archive inactivity. We examine these possibilities in the next section.

6.1 Space savings through differences and compression

We examined the effect of applying two optimizations to stored objects:

- Compressing objects.
- Representing new versions by storing differences with previous versions (instead of as whole files.)

We used the Unix `gzip` utility to perform the compression, and the Unix `diff` utility to compute the differences. We calculated the difference over only the file contents, not the metadata (e.g. timestamp, etc.) To retrieve the original objects, we apply `gunzip` to decompress, or `patch` to construct a whole object from differences. We examined applying these optimizations separately and together; when applying them together, we calculated the difference first and then compressed objects.

Because the data set reported above had so few file modifications (14) we collected a new data set to perform these experiments. This new set was collected from the filesystem over a four day period, and over a larger portion of the filesystem. There were 86,694 unique files collected, and 2,807 file modifications. The archive had a total of 11.26 Gb of data.

The results are presented in Table 4. As the table shows, the compression achieves the most space optimization, reducing the size of the archived objects by 39 percent. Although the time to compress each file was only 0.093 seconds (about 0.73 second per Mb), the total time was quite large (more than two hours). This result suggests that the compression is best done in the background or lazily if many objects are added to the archive at once.

The space savings from the difference optimization was quite small (1.2 percent). Although differences can save significant space per file, the number of "new versions" was small compared to the number of "original versions." Since the original version must be stored whole, the aggregate effect of the difference optimization is small. In a system where there are many updates to files, the difference optimization may become more important.

7 Extending the InfoMonitor's capabilities

In the discussion so far, we have assumed that the InfoMonitor archives relatively static content. Files such as HTML pages may change over time, but between changes pages exist in a stable state on the filesystem. We have also assumed that we have access to the filesystem used by a web server. In practice, one or both of these assumptions may not hold. We may wish to archive "dynamic" content, or content that is generated on the fly as users contact the web server. We may also wish to archive web sites run by other organizations, to which there is only an HTTP interface. In this section, we discuss how we might extend the InfoMonitor to manage these situations. In each case, the basic architecture of the InfoMonitor remains the same, and it is only the interface to the data source that must change. This extensibility is one of the ways in which the InfoMonitor improves upon the traditional file backup model.

7.1 Dynamic content

Much of the web's content is generated dynamically. Users make requests, and pages are generated on the fly to respond to these requests. Examples include weblogs, interactive map services, online travel agencies, and so on. In these cases, the basic two tiers of HTTP server and filesystem are augmented with the following components:

- A database system for storing information that forms the dynamic content. Frequently, this is a relational database system.
- Application logic for generating dynamic pages from the database system based on user requests. This logic may be encapsulated in simple CGI scripts, or may be a standalone middle-tier application server.

The consequence of these additions is that dynamic content does not exist statically as HTML pages. In this case, we can archive this content by effectively archiving the database tables and the application logic. This will allow the dynamic pages to be reconstructed from the archive if necessary by creating a new instance of the database, and running the application logic on this new instance. (See Section 7.2 for the case where we cannot access the database and application logic directly.)

There are two options for archiving the database system. One option is to utilize the native archiving features of the database system. Unlike filesystems, many database systems are designed with archiving capabilities, and we can use these capabilities directly. If the system has no such features, or they are not appropriate for our needs, the second option is to use the InfoMonitor to extract information from the database for archiving. For example, the InfoMonitor could query the system for changes (e.g., using the techniques in [21]). We can then treat changes as new versions, and manage these versions in the same way we dealt with file modifications.

We can archive program logic in a straightforward way. Since CGI scripts, Enterprise Java Beans, PHP scripts and other application components are usually stored as files on a filesystem, we can archive these files in the course of normal InfoMonitor operations. However, we may not be able to run these program modules when we retrieve them from the archive. Programs and scripts are written or compiled for a specific architecture and execution environment, and if this environment is not available when the program is retrieved from the archive, it cannot be run. This problem is partially ameliorated by platform-independent languages such as Java (and to a certain extent Perl and other scripting languages). However, even these languages require an interpreter or virtual machine for the correct language version to be available for the machines of the day. Moreover, since we assume "non-intrusive" archiving, we cannot dictate that website designers use a particular language simply to facilitate archiving. Solving this problem in the general case is difficult, and researchers are only beginning to make progress (see for example [22]).

Once the InfoMonitor can use an appropriate interface for extracting database information and application logic, it can archive these objects in the same way it manages static files. The basic architecture of the InfoMonitor as a bridge between a data source and an archive is retained.

7.2 External web sites

If we want to archive a web site run by an external organization, we cannot access the filesystem or underlying

	Time (to compress or calculate difference)	Total archive size
Base (no optimizations)	-	11.26 Gb
Compress only	2 hours 19 min. 40 sec.	6.88 Gb
Difference only	7 min. 4 sec.	11.12 Gb
Compress and difference	2 hours 19 min. 4 sec.	6.70 Gb

Table 4. Results from compression and difference optimizations.

database directly. Instead, we can only access the HTTP interface to the site. By building a component that utilizes HTTP to extract the web site's content, we allow the InfoMonitor to treat the pages in the same way that it treats files; that is, as a series of version chains representing the changes undergone by individual pages.

If the external web site contains static content, we can crawl the hyperlink structure to find all of the pages served by the web site. For now, we assume that all of the content of the site exists within one domain (e.g. `stanford.edu`) and is reachable from the root page of the site. The first time we archive the site, we can start at the root page and crawl until we have discovered all of the pages at the site. Links to pages outside the domain are not followed.

After the initial crawl, we must discover new pages added to the site and new versions of existing pages. We can restart the crawl at the root, and follow the hyperlink structure. In this case we discover new pages and new versions of existing pages as a part of the crawl. However, restarting the crawl requires retrieving and examining every page in the site, an expensive process (similar to the slow scan).

A more efficient alternative would be to emulate the quick scan by retrieving just the timestamp or other information indicating that the page has changed, without retrieving the whole page. The HTTP 1.1 protocol defines a request type called "HEAD" which allows the crawler to retrieve metadata about the page, such as "Content-length" and "Last-modified," without retrieving the whole page. This metadata allows the crawler to detect new versions of existing pages; the crawler can then use a "GET" request to retrieve the whole page if it is a new version. Newly created pages must be linked to by an existing page, and this existing page must have changed to incorporate the link to the new page. Thus, when the crawler retrieves a new version of an existing page, it should parse the page to find links to any new pages. In this way, the crawler can efficiently retrieve both new versions and new pages. More techniques for efficiently crawling are described in [4].

We must also deal with the case where we want to crawl more than just one physical web site. That is, we want to crawl a "logically coherent" collection of pages that may span several machines, or may form only a subset of a larger domain. For example, `acm.org` contains the website of the ACM as well as a mirror of the DBLP[3], a bibliography web site. Imagine that we wanted to archive just the DBLP pages. Because the DBLP contains links to the ACM site proper, the rule "do not leave `acm.org`." is not sufficient. If the crawler starts at the root DBLP page, it will eventually follow a link to a non-DBLP ACM page, and thus will find and archive the entire ACM site. Therefore, the crawler must have more expressive rules for determining the extent of its crawling and archiving activity.

Finally, the most difficult challenge is to archive dynamic content residing at an external web site. In this case, we cannot access the database or application logic directly; the only access is via HTTP. Dynamic pages may be generated as a result of a form request, or they may be generated automatically but accessed using the same hyperlink process as static pages. In the case of form-based dynamic content, it is possible to extract all or part of the accessible content by automatically filling out forms. For example, Raghavan and Garcia-Molina [24] describe techniques for "guessing" the appropriate values for form fields. In the case of hyperlink-accessed dynamic pages, the crawler can retrieve pages in the same way as static web pages, but may need to crawl more frequently since dynamic pages change more frequently. If the dynamic page is always changing (e.g. a new page is generated for every request), then it is impossible to retrieve all versions of the page. In this case, the InfoMonitor administrator must decide how many instances of the page should be archived to form a representative sample.

8 Related work

Considerable work has been done in the area of increasing the reliability of traditional filesystems. Some of this work has focused on data backup. A survey of current backup techniques is presented by Chervenak et al in [3], and Hutchinson et al [15] discuss the nature of the backup snapshot. Traditional strategies often require human intervention to mount magnetic tape, run the backup, and store backup tapes in a coherent way. Recently, systems designed to better automate this process have been introduced by IBM [6,16] and UniTree Corporation [17]. King et al [19] suggest that a remote system storing backed up data could provide quick recovery in the event of a

[3] `http://www.acm.org/sigmod/dblp/db/index.html`

crisis. We discuss differences between the InfoMonitor and backup systems in Sect. 2.2.

Other work has focused on making the primary medium itself more reliable. Patterson et al argue for Redundant Arrays of Inexpensive Disks (RAID) in [23], and Schloss and Stonebraker extend this idea using distributed disks in [28]. Several researchers have proposed using logs as an integral part of filesystems to increase reliability; these include Rosenblum and Ousterhout's Log-Structured Filesystem [26]. Goldberg et al [12, 2] have designed and implemented an "Archival Intermemory" system that models a reliable archive as a RAM-like data store. Each system assumes the filesystem can modified significantly to incorporate the reliability features, which may not be possible. Moreover, these systems do not address the problem of integrating components with different properties; instead, the feature set of these reliable systems resemble that of a filesystem.

The data store integration problem is a common one in data warehousing. Researchers have investigated the problems of designing the architecture of warehouses, detecting changes in source data and maintaining view consistency [20, 21, 1, 30]. We discuss the relation of data warehouses to our work in Sect. 2.2. Layered filesystems are examined by Khalidi and Nelson [18], although they assume that filesystems contain hooks to facilitate layering.

Archiving data, including legacy data, is a significant challenge facing the digital library community. A survey of the problems and possible solutions is presented in [11], and the implications for solving archiving problems are discussed in [27]. Many digital library archiving schemes circumvent the component integration problem by migrating data to a reliable archive and providing access as a service. Examples include the Computing Research Repository (CoRR) [14], and the collections being built by the San Diego Supercomputer Center [25]. These schemes do not allow direct access to the original data store, which could be important to minimize the impact on users.

Many web sites use tools to author and manage the web pages, and it may be possible to integrate versioned archiving into these tools. For example, Microsoft's FrontPage package [8] provides RCS-like versioning of documents. However, current tools do not provide archiving at the level provided by the SAV system, and may not even provide any backup capabilities at all. Moreover, a robust solution should be applicable to web sites that do not use a central authoring system, allowing different users to use different HTML editing packages.

9 Conclusions

In this paper we have presented the InfoMonitor, a system that increases the long term reliability of data objects, even if those objects are not already stored on a reliable archive. This problem is especially apparent in the context of a web server's unreliable filesystem. Although this filesystem is convenient for users, an archiving component must be added to provide long-term reliable storage. The InfoMonitor demonstrates how the filesystem can be unobtrusively archived, preserving information without disrupting users. The InfoMonitor must bridge the gap between the features of the filesystem and the reliable archive. We have discussed how the system detects data modification events, deals with naming, and finds members of a logical set. We have also explored the performance of the InfoMonitor, examining the scalability to large data sets and studying improvements which could further optimize the system. The InfoMonitor deals with the specific situation of a web server, but the principles it illustrates can be applied to a wide variety of data storage systems whenever it is desirable to construct additional layers to preserve data.

References

1. S.S. Chawathe, A. Rajarman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.
2. 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.
3. Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings Joint NASA and IEEE Mass Storage Conference*, March 1998.
4. Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Conference on Very Large Databases (VLDB)*, September 2000.
5. Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive. In *Proceedings of the 4th European Conference on Research and Technologies for Digital Libraries (ECDL)*, September 2000.
6. IBM Corporation. Adstar distributed storage manager (ADSM) - distributed data recovery white paper. http://www.storage.ibm.com/storage/software/-adsm/adwhddr.htm, 1999.
7. Inktomi Corporation. Web surpasses one billion documents. http://www.inktomi.com/new/press/-billion.html, 2000.
8. Microsoft Corporation. Microsoft FrontPage. http://-www.microsoft.com/frontpage/, 2000.
9. Arturo Crespo and Hector Garcia-Molina. Awareness services for digital libraries. In *Lecture Notes in Computer Science*, volume 1324, 1997.
10. Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries*,

1998. Accessible at http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1998-0082.
11. John Garrett and Donald Waters. Preserving digital information: Report of the Task Force on Archiving of Digital Information, May 1996. Accessible at http://-www.rlg.org/ArchTF/.
12. Andrew Goldberg and Peter Yianilos. Towards an archival intermemory. In *Advances in Digital Libraries*, 1998.
13. Anja Haake and David Hicks. Verse: Towards hypertext versioning styles. In *Hypertext '96*, 1996.
14. 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.
15. Norman C. Huchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.
16. Tivoli Systems Inc. Tivoli storage manager. http://www.tivoli.com/products/index/storage_mgr/, 1999.
17. UniTree Software Inc. Unitree technical overview. http://www.unitree.com/overview/overview.htm, 1999.
18. Yousef Khalidi and Michael Nelson. Extensible file systems in spring. In *Proceedings 14th Symposium on Operating Systems Principles*, December 1993.
19. Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *TODS*, 16(2):338–68, 1991.
20. W. J. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. In *Proceedings of the International Conference on Data Engineering*, April 1997.
21. Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.
22. Raymond A. Lorie. Long term preservation of digital information. In *Proceedings of the 1st Joint ACM/IEEE Conference on Digital Libraries (JCDL)*, June 2001.
23. David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.
24. Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proceedings of the Conference on Very Large Databases (VLDB)*, September 2001.
25. Arcot Rajasekar, Richard Marciano, and Reagan Moore. Collection-based persistent archives. http://www.sdsc.edu/NARA/Publications/OTHER/-Persistent/Persistent.html, 2000.
26. Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings 13th Symposium on Operating Systems Principles*, 1991.
27. Jerome H. Saltzer. Technology, networks, and the library of the year 2000. In A. Bensoussan and J.-P. Verjus, editors, *In Future Tendencies in Computer Science, Control, and Applied Mathematics. Proceedings of the International Conference on the Occasion of the 25th Anniversary of INRIA*, pages 51–67, New York, 1992. Springer-Verlag.
28. G.A. Schloss and M. Stonebraker. Highly redundant management of distributed data. In *Proceedings of Workshop on the Management of Replicated Data*, pages 91–95. IEEE, IEEE Computing Society, November 1990.
29. Walter Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.
30. Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.
31. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.