Technical Report CS-94-35, Department of Computer Science, University of Virginia

An Archive Service with Persistent Naming for Objects

John D. Jones, James C. French

Technical Report CS-94-35
Department of Computer Science
University of Virginia
August 1994

Abstract

Wide-area systems for information storage and retrieval are rapidly gaining in popularity. Examples include FTP (File Transfer Protocol), Gopher, and World Wide Web (WWW) archives of many types of information. Each system provides a means of naming a file or data object so that others may retrieve the information. Unfortunately, this name depends on the network address of the server and on the filename within that machine. Some flexibility is gained by using aliases for these names, but problems persist. Additionally, the use of aliases does not handle the replication of files at several sites, or the movement of a file from one site to another.

The result is that these names frequently become invalid. A name that is good today may not work in a week or a month. If a name for a useful collection is passed on to others, it may not work by the time they try to use it. For these reasons, a better approach to naming is needed. In this paper we present a prototype distributed service for archiving files. Each file is given a permanent name, which can be used to retrieve the file from any site of the archive. If the contacted site does not have the data, it returns the current location, which is accessed automatically by the client. This allows files to be moved and replicated within the archive without invalidating any of the names. In a hypermedia environment such as the WWW, this means that the links will remain valid, avoiding the need to change each document that references this data.

We have developed this system to demonstrate that this approach is feasible. We describe both a standalone service that is accessed via a command-line client, and a gateway to the service from the World Wide Web. The gateway allows the archive to be accessed from NCSA Mosaic, or any client that can use the Hypertext Transfer Protocol (HTTP).

The service is assumed to be global in scope, so issues of scalability are critical. This influenced our design, and led us to experiment with caching of names, and automatic replication of frequently accessed files. We have, however, omitted the implementation of a distributed name resolver, using a centralized one instead. A distributed implementation would be required for an actual service, and we describe how this might be built. The task of locating an archive site is also discussed.

Technical Report CS-94-35, Department of Computer Science, University of Virginia

An Archive Service with Persistent Naming for Objects

John D. Jones, James C. French
University of Virginia

INTRODUCTION

Wide-area systems for information storage and retrieval are rapidly gaining in popularity. Examples include FTP (File Transfer Protocol), Wide Area Information System (WAIS) , and World Wide Web (WWW) archives of many types of information. However, as described in the next section, these systems break down when a server or data object is moved. This causes the names to become invalid, and links that use them will fail. Users need a system that provides a reasonable guarantee that the data will remain accessible.

To solve this, we propose a location-independent naming scheme. The names are permanent (valid for the life of the object and unusable thereafter) and resolved, as necessary, by a name-to-location lookup. This allows files to be moved and replicated within the archive without invalidating any of the names. In a hypermedia environment such as the WWW, this means that the links will remain valid, avoiding the need to change each document that references this data.

To create user trust, the service guarantees that the data will always be available from any access point when its name is provided, and that the data will not be modified if the data is stored with an immutable object type. We have developed a prototype service to demonstrate that this approach is feasible. We show how to use a simple set of primitives to implement a complete set of functions. We demonstrate a gateway from the World Wide Web, and discuss caching of names, and replication of frequently accessed files.

All elements of the service are distributed and scalable, with the exception of the name resolver. A distributed resolver would be required for an actual service, and we describe how this might be built. Other approaches to persistent, location-independent naming are discussed in the “Related Work” section.

BACKGROUND

Many systems exist for providing information electronically to the global community. Older systems include FTP archives and data available on request via electronic mail. More recently, menu-driven, search-based, and hypertext systems such as gopher, WAIS, and WWW have become popular.

Electronic information retrieval is gaining popularity due to a number of factors, including:

- a growing number of users on the Internet
- the increasing availability of hardware and software to support the retrieval and display of images, audio, and video, in addition to traditional text
- growing collections of information available online

Another major factor is the wide availability of software, including browsers for the WWW

which combine the diverse sources and multiple protocols into a single easy to use system. The Web accomplishes this in two ways. First, its browsers (for example NCSA Mosaic) incorporate a large number of protocols. Second, and at least as important, a naming scheme is used that encompasses all of these protocols. This allows a single name (albeit with several fields) to be built, saved, transferred, and used in hypertext documents. Thus indices and directories of Internet resources are written, making documents, software, and other data accessible with the click of a mouse.

THE PROBLEM

However, the names in the WWW[2], and all major systems currently in use (including WAIS [12] and the others listed above) employ a naming scheme that identifies a specific copy of a resource, on a specific computer. Domain name aliases (in the Internet) and soft file links (in Unix) allow administrators some flexibility in changing hosts or renaming files. However, such approaches do not handle arbitrary file movement nor multiple copies of frequently accessed objects.

Many events may cause an Internet resource to be moved or replicated. When a popular source comes on-line, it often receives enough attention to swamp the host it is funning on. To handle this, the administrators may move it to another machine with more power, or where it will not interfere with local users. When this happens, any existing links to the site are broken. There is no effective way to update all the pointers, as they may have been placed in many indices and hypertext pages, saved by individual users, even given from one user to another. The same problem occurs whenever an Internet resource is moved, although it may, in some cases, be dealt with by defining the name of the old host as an alias for the new, if it is considered worth the trouble.

To handle a continuous load for a very popular resource, however, it is necessary to make copies of the resource, preferably geographically distributed. As above, it is not feasible to update all copies of the name when the site is replicated. But in this case, it is not even possible to build a name that reflects all of the copies. Trying to distribute access from users among the copies is sometimes done by providing a list of names from which to choose. However, in many instances, just a single name may be kept, and access can be very uneven.

To address this problem, a number of Internet-wide indices have been built to allow users to search for a resource, rather than retain a pointer to it. Examples include Archie (for FTP archives) and Veronica (for Gopher sites). However, finding information using these methods is inexact (usually based on keyword matching on file and directory names), requires user interaction to sift through the possibilities, uses substantial resources to continually rebuild the index, and does not guarantee that the same resource would be found by two independent searchers.

OUR APPROACH

To address the problem of persistent naming, we have designed and built an Archive Service that assigns permanent, location-independent names to objects. When an object is stored, the Service generates a name, records it in an internal name-object database, and returns this name to the user. Subsequently, on a retrieve request, a user may present this name to any site in the Archive Service. If the object still exists, the Service will provide it to the user.

If the object is stored locally at the site which the user first contacted, the data is returned immediately. Otherwise, this site consults the name-object database to find the current location of the object. In order to avoid additional network hops (when the data is closer to the user than the Archive site is), a referral message is returned to the client, giving the location. In order to provide a seamless retrieval to the user, the client extracts the correct location from the referral, and contacts this site to obtain the data.

One goal of the Archive Service is to provide users with a meaningful guarantee on the semantics of a name. Users should be able to rely on the fact that an Archive Name represents a known and reproducible data object. To provide this, an object may be declared as immutable. Users can retrieve the characteristics of an object to learn that it is immutable, and the Service will enforce the lack of changes. In this way, a user may be certain that retrievals based on a given name will always yield the exact same data, independent of when or by whom the retrieval is initiated. For reasons to be described later, mutable objects are also allowed, and several characteristics are maintained for each object.

To make the service simple and keep the interactions clear, we defined set of basic primitives, which the service provides. These are the only functions that the Service provides to users:

- store -- add an object to the Archive.
- retrieve -- given a name, get an object that is stored in the Archive.
- delete -- completely remove a name and attached data from the Archive. *Not Implemented.*
- update -- replace the data attached to a name with a new set of bytes. *Not Implemented.*
- describe -- return basic information about an object, such as size. *Not Implemented.*

We believe that these primitives are sufficient to provide the guarantees necessary for names. To demonstrate this, we describe how the primitives can be used to build other features that are often desired from a persistent naming scheme. We view these features as necessary for user convenience, but not part of the basic concept of a location-independent name.

- get latest version -- e.g. of a weather map or stock market report
- retrieve characteristics -- may go beyond the simple attributes maintained by the Service.
- retrieve all items in a collection

These functions can be implemented by adding an extra layer on top of the Archive Service. The layer would act as a client and can therefore only interact with the Service via the primitives listed above.

In this document, we use the term “name” to refer to the location-independent name that identifies an object as distinct from other objects. We use the term “location” or “location string” to mean the location-dependent identifier of a copy of the object. A “location” consists of both the name of the host where the copy is stored, and the local file name containing the data.

RELATED WORK

Other approaches to persistent, location-independent naming are being pursued.

IETF

The Internet Engineering Task Force (IETF) has done much recent work to define what is needed for names in the Internet. This covers the syntax [1], [2] to be used, requirements for location independent names [3], and scenarios of how additional characteristics may be used to assist in finding names [4], [5], [6]. IETF research is published in Internet Drafts available via anonymous FTP. Additional insight, and discussion of research not yet reported in Internet Drafts is found in the mailing list for the Uniform Resource Identifier (URI) Working Group [7].

URN

The Uniform Resource Name (URN) is the primary persistent, location- independent name under consideration by the IETF. They envision that the URN will eventually become the standard mechanism to identify a resource. While the URL identifies the current location of a resource, the URN is meant to capture the meaning, so that, if the resource moves, a user with a URN will still be able to retrieve what is wanted. [3] Accordingly, this will sometimes require a name-to-location translation, using a lookup service. For this, the IETF is currently proposing to use Whois++. [5]

However, assigning the URN to identify the meaning of an object opens the door to many interpretations. To some, this is merely a specific collection of bytes, verifiable by checksum. For others it may include all formats of an object (such as postscript or plain text, in the case of a document) or perhaps the latest version. The current direction is to allow the URN to encompass all these things and more. There is also a desire that users will be able to determine whether two objects are -- by any such definition -- the same.

Accordingly, the semantics of a name is deliberately and explicitly left undefined. In the same way that the generation and assignment of names is delegated to hierarchically organized naming authorities, each authority is solely responsible for the meaning of the names it assigns. Whether two objects are “the same” (so that they can share a name) is seen as a local decision. [3][8]

This makes the URN a more heavyweight concept than the names we propose in the Archive Service. In addition to supporting all the above mentioned meanings, a URN should be resolvable into a wide range of attributes, called Uniform Resource Characteristics (URC). The definition of these is similarly open-ended, including “ANY conceivable type of meta-information or URI.” (URI, or Uniform Resource Identifier, is a generic term encompassing URN, URL, LIFN, etc.) [6] [5] This allows the characteristic information to go far beyond the basic set maintained by the Archive Service. As a URC may include the author’s name, address, a title, description, instructions for use, etc., it can contain much more data which must be stored by the server providing the object, and possibly by the server resolving the name.

Another result of the flexibility of the URN is that it provides the user no guarantee that a name has a specific meaning, that it can be used to retrieve exact object the user wants (if, for example, it has been updated and an earlier draft is needed), or even that two users who ask for the same

object (by name) will get the same data. A particular naming authority may provide such a guarantee, but it will be difficult for users to know the policies of all name authorities or keep track of them.

LIFN

To address this, in the IETF discussions, the LIFN is proposed as a name that uniquely identifies a specific object. It goes to the point of identifying a particular, immutable stream of bytes. To ensure this, there are proposals to encode a checksum signature into the LIFN so that users can determine whether the data was changed. [9]

The difference between this and names in the Archive Service is that there are no mutable names. Mutable names are needed in the Archive Service to allow support for URN-like functionality built on top of the Archive Service. Since the IETF proposes implementing URNs outside of LIFN-sealed objects, they don't need a mutable object type. We feel that mutable objects in the Archive Service allow for a clean addition of a URN-like layer, taking advantage of the work already done to store and retrieve data in a distributed way. Providing an immutable object type for other objects allow users to name objects that they can be sure will not change.

Whois++

Whois is a "netwide directory service." At many sites throughout the Internet, databases are maintained in this and a variety of other formats, providing "information about individuals, departments, and services." Whois++ is an effort by the IETF to combine the best features of these services and produce a standard for Internet directory services. Three types of records are considered in the basic service: people, hosts, and domains. [8]

Extensions consider its use as an indexing and directory service for all resources in the Internet. [9] The indexing would use registered database names (rather than host names), helping users locate the correct Whois server, and protecting the names from being invalidated when a host is moved. A more structured format and record types are also introduced, to support machine reading of the information returned. However, only the interface for queries is defined, encouraging servers to experiment with improved search techniques.

SOLO Protocol

Experimentation has been done at Inria on a white pages server using the Simple Object Lookup protocol (SOLO)[10]. The data tables are inspired by X.500 records, but implemented using the same approach as Whois++. The service uses the white pages lookup to find a document. To make it easy to locate and retrieve documents, they implemented a gateway to the WWW and modified a Web browser (actually, they changed the Web common library, *libwww*, which is linked into many browsers) to connect directly to a SOLO server with a URL containing a new protocol specifier, "solo:". [11]

This integration of a new name type into a Web client is something we envision as necessary for the Archive Service to eliminate the hostname from the URL and make a truly location-independent name. This is also implicit in the IETF plans for URNs, where clients would be able to accept names of the approximate form, "urn://naming-authority/assigned-name". Basic names for the Archive Service might appear similar to "archive://archive-name", reflecting the flat name

space. Eliminating the host from the name moves the selection of a server from name-creation time (after which it may become stale) to client execution time, allowing more flexibility, and the chance for different clients to connect to different servers, spreading the load out better.

Jade File System

Jade is a file system designed to be Internet-wide. As a single global hierarchy would be too unwieldy to navigate, Jade allows each user to build a personal directory tree, mounting remote physical file systems at the points they specify. To identify remote files, Jade uses a naming scheme (primarily in order to mount them) that includes the host and local file names. To be useful as a user's basic file system, Jade employs file caching and other mechanisms to improve performance. [13]

For location independence, a directory structure is tied to a user, not a specific machine. When a user logs in, the appropriate directory information is retrieved and used to locate the actual files. However, the canonical name of an objects contains a physical host name. For two users to share a name, they must communicate with the location dependent name (or use an indirect name, using the sending user's login name and the file name within his space). Further, if the remote file system is transferred to another host, the old name becomes invalid.

Cedar File System

Cedar is an approach developed at Xerox PARC during the 1980s. It also uses names that indicate the server where a file is stored. As the guiding model was of a team of developers needing shared access to program files, a limited number of servers was envisioned, and a simple name is used for each. [14] This does not support a global repository, although it could easily be extended by adding domains to the server names.

The system is interesting in that it only supports sharing for immutable files, avoiding the cache consistency problem. Changes to files are handled by assigning new version numbers for updates. It is always possible to open a precise copy by specifying the version number in the name. However, a read may also request the latest version of the file, gaining some of the advantages (and problems) of mutable objects.

Other Distributed File Systems

Conventional distributed file systems are surveyed in [15] and [16]. These, like Jade and Cedar, are designed to be the primary filesystem used to store and retrieve files for everyday use. Commercial and experimental file systems in this category include Sun Network File System, Andrew File System, Sprite, and Locus.

These systems typically employ a naming scheme that includes the storing host, making the names location dependent. This is not always true, however. For example, Andrew uses location independent names that are mapped to servers in a fully replicated "location database." Andrew makes this less onerous by mapping at the directory subtree level. A similar approach is used in Locus. This does not seem feasible as a resolution approach for Internet-wide archives, but the name service could be reimplemented in a more distributed fashion, without needing to change the names themselves. This would avoid affecting the other parts of the system (which use the names).

ARCHITECTURE

The architecture of the Archive Service is described in this section. Each of the major components is described, and implementation choices are explained. Additional features and examples of use are included in later sections.

Implementation

The Archive Service is implemented as a set of five server modules and a command-line client.

The server modules are:

- an Access Point, which is the Service's interface to users
- a Name Generator, which provides globally unique names for objects
- a Physical Storage module, which saves the data into disk files
- a File Manager, which performs automatic replication of files to improve performance
- a Name Resolver, which maintains a list of all objects and maps names to locations

The purpose of the client is to format request messages, and send them to the Access Point. A single site, illustrating the internal parts, is shown in Figure 1. Arrows show the direction of possible requests.

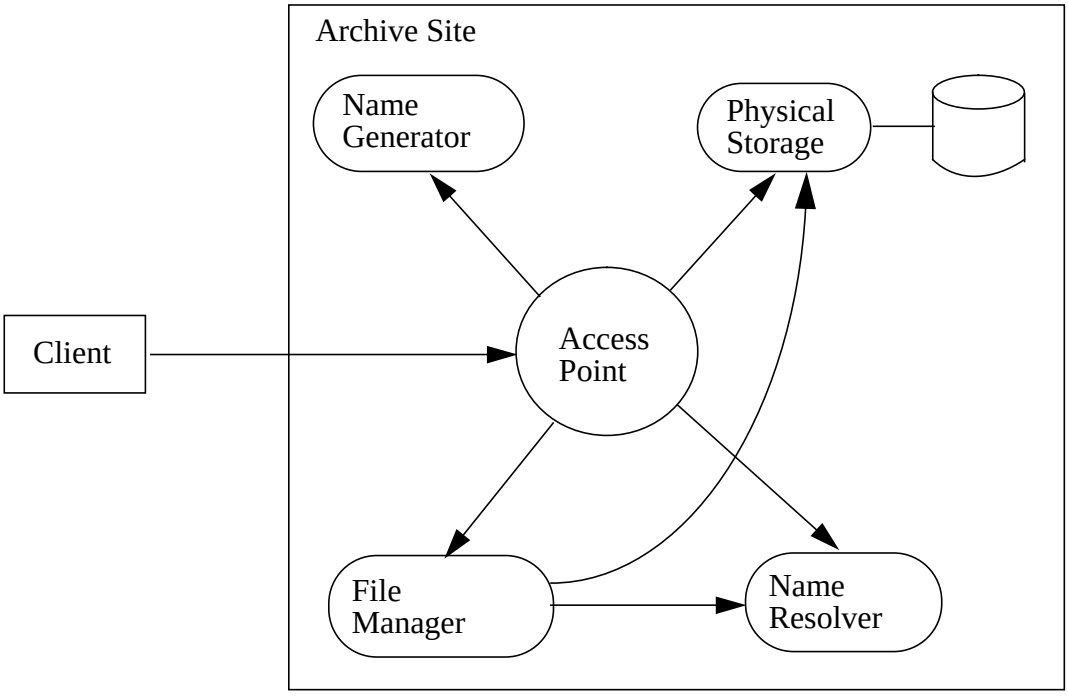

Figure 1: The Components of an Archive Site

The server modules represent a logical breakdown of the functionality. We implemented these as separate processes in order to simplify the development. Because we did this, the interactions

between the modules are simple and well defined. A possible optimization would be combining many or all of the modules into a single process, replacing the request-response messages with procedure calls. To help support this, the internal structure of the Archive Service is hidden from the user who only interacts with the service through the Access Point.

The overall design for a full Archive Service contains many Archive sites, each with an Access Point, Name Generator, Physical Storage, and File Manager. However, in the current implementation, the Name Resolver is centralized, residing at just a single site. When name resolution is required at another site, the module that needs service opens a socket to the Resolver on a remote host. We recognize that, in a scalable service, this component must be distributed. Therefore, the design can accommodate any number of Resolvers, and under “OPERATIONAL ISSUES” on page 29,we discuss how to maintain the name space using a distributed implementation. A Service-wide view, illustrating multiple sites and a central resolver, is shown in Figure 2.

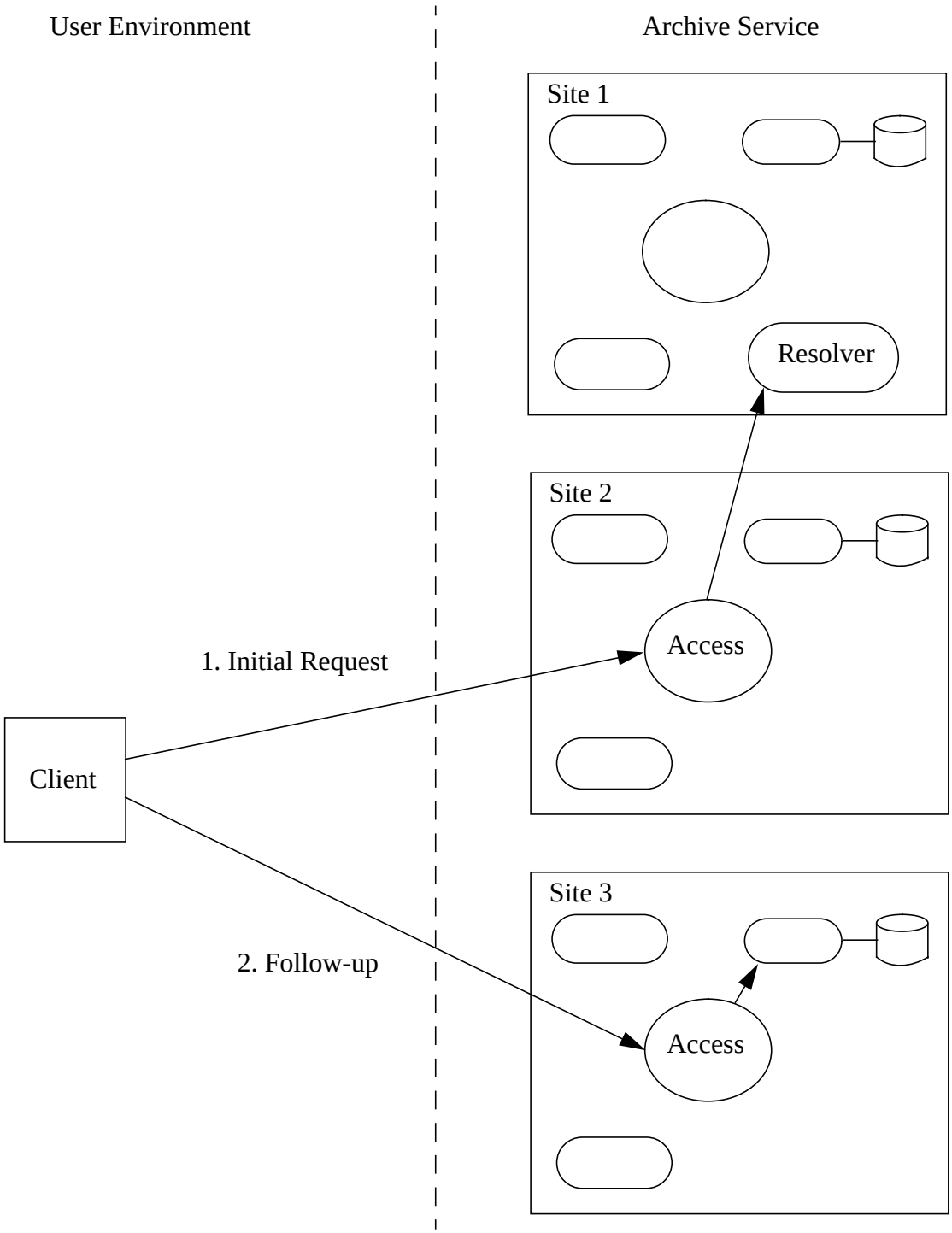

Figure 2: Many Sites Using a Central Resolver (Retrieval shown)

Arrow 1 is an initial Retrieve request to any site. After this returns a Referral message, arrow 2 shows a follow-up request to the site where the requested data is stored. See the section on the

Retrieve primitive for more information on referrals.

Access Point

The Access Point is the user's only entry point into the Archive Service. It coordinates the execution of user requests, by generating the necessary requests and connecting to other servers on the user's behalf. The Access Point would be responsible for enforcing security by ensuring that users only perform operations they have permissions for.

Clients may connect to any Access Point, but are assumed to use one that is near to them, or one that they have found to be lightly loaded. Because an Access Point must block and wait for a response after connecting to another server, it spawns a child to handle each user request. The child process executes the request, returns the response to the user, then exits.

Name Generator

The Name Generator is the simplest part of the Service. Its sole purpose is to provide a globally unique name whenever requested to do so by the Access Point. The current name format is similar to numerical Internet addresses. An Archive Name also has four integer fields separated by dots, so that 190.11.103.5 could be assigned. However, each field is designed to be stored (internally) as a four byte value, so it may have a maximum value of 2^32 - 1, for a total of 2^128 possible names (about 3 x 10^48).

In our implementation, a distributed Name Generator can still guarantee that each name is unique because each site is configured with a unique generator id, which is used as the first field of the name. This allows for over 4 billion Archive sites and about 8 x 10^28 names per generator.

Name Resolver

In the current implementation, a single Name Resolver serves all Archive sites, translating names to the physical locations where the data is stored. Each server module needing name to location mapping must know the address of a Name Resolver. As an independent module, however, the Resolver is replaceable in case a better name server is implemented, as discussed under "OPERATIONAL ISSUES" on page 29.

When multiple copies of an object are made, the location of each is stored in a Name Resolver. To service a Retrieve request, a location must be chosen from this list. First, the Resolver checks for a copy on the same host as the Access Point, and returns it location if one is found. Otherwise it returns the next copy in the list, using a round-robin policy on all locations for the object.

Name Locking: Integrity of a data object is ensured by requesting a Name Resolver to place a lock on the name before the object is modified. If any other Lock requests or other operations that modify the object (Delete Copy, Delete Name, or Replace Locations) arrive while the name is locked, the Name Resolver will hold them by placing the request and the socket it arrived on into a queue attached to the name. When the name is unlocked, the Name Resolver takes the first request off the queue and executes it, sending a response back across the socket.

There are some consequences of this approach. First, if there are many Update or Delete requests and the queues build up in the Name Resolver, then that process may have too many sockets open

and exceed the maximum allowable number of open file descriptors. This may not be limited on some systems, but in our testing, we found this limited to 64 by default. Second, if the site handling an Update request crashes while it has the name locked, the name will become stuck in the locked state. A mechanism for correcting this is required. The most straightforward approach is to use a log at the updating site. *Despite the above description, note that name locking has not been implemented.*

Physical Storage

The Physical Storage module is responsible for saving the Archive data on stable storage. In the current implementation, this is a local disk on the host it runs on. This is not truly stable storage, as it is not duplicated, but it is close enough for this prototype. The process reads the location (path) and maximum size of the storage from the site's configuration file.

The Storage server waits in a loop for socket connections to come by and when they do, it performs one of the following tasks:

- Storage of data: It returns an ack if it was able to store the file onto the secondary storage. It could return a failure message if storing the file would mean that the disk quota would be exceeded.
- Retrieval of data: It returns the file if it was able to find it and read it.
- Update of data: It returns an ack if it was able to store the new copy onto the secondary storage. It could return a failure message if storing the new copy would mean that the disk quota would be exceeded.
- Deletion of data: It returns an ack if it was able to delete the file from the secondary storage.

File Manager

The other complex piece in the design besides the Name Resolver is the File Manager. A File Manager process at each site bears the responsibility for implementing the File Placement algorithm. If enabled, it runs this algorithm periodically to determine what changes must be made, then connects the Name Resolver and Physical Storage modules to effect the changes.

To accomplish this, it maintains its own table of names, recording only those that have been accessed recently or are stored locally at the site. It is notified by the Access Point whenever files are added, deleted, or requested on the local system. To allow for manual file placement, the File Manager may be disabled, so that it will not run its algorithm.

Client

We developed a command line client that runs from a Unix shell on the same platforms as the Archive servers. It allows users to store and retrieve objects or send a control message to an Archive server. Sending the control message makes this an administrative tool in addition to a client for users. The interface is described under "INTERACTING WITH USERS" on page 25.

Common Libraries

Message library: The message library abstracts out the socket connections and messages formats. We wrote routines to parse individual fields or an entire message into an internal data struc-

ture. Other routines translate an internal structure into a properly formatted message. On top of these we built routines that format and send a message, even some that open a connection to a remote server, send a request, wait for the response, and evaluate the response type.

Error routines: We devised our own error routines for diagnostic messages. The error reporting system also included a method for logging events happening in the system so as to track the execution. The log level is a parameter that can be specified in a configuration file.

Sockets Library: We borrowed a C++ sockets library from our colleague Charlie Viles. This class provided a simpler abstraction for Berkeley sockets by performing low-level operations and choosing useful default options.

Name Table Routines: Because the Name Resolver and File Manager modules each keep a table of names, code is shared between them. This code maintains a hash table indexed by name. The Name Resolver uses this to keep a list of all the locations for an object, while the File Manager keeps usage data for files stored or requested locally. Unfortunately, because the tables in the two modules are not identical, the code was cloned rather than abstracted, so there is a special copy for each of the modules.

Object Types

The Archive Service supports the following object types and characteristics. This minimal set is needed to perform functions such as storage management and security (only allowing the owner to delete an object). *Note that, as Update, Delete, and Describe are not implemented, these are not currently enforced or visible to users.*

- Mutable/Immutable
- Deletable/Permanent
- Owner (Storing User)
- Size in bytes

Message Protocol

A messaging protocol was defined to allow client and server processes to communicate requests and responses for the operations and primitives described above. The available requests and the expected return messages are described in the Fundamental Operations section. A complete list of message types and the fields included in each is given in Appendix 1.

To begin a request, the client opens a socket connection to the server and sends the message across. For user-invoked clients, the server must always be an Access Point. However, any of the servers may themselves act as a client and make a request of another server. After processing the message, the server returns a reply message and closes the connection. This reply may be an acknowledgment of successful completion, a message carrying the requested data, or a failure notification.

All messages begin with a header, which consists of a variable number of fields, depending on the message type. For a particular message type, however, a specific set of fields is required. All possible fields are assigned a position in the message (shown in Appendix 1), and the field may only appear at that point, thus fixing the order of fields in a message.

Most fields consist of one line, starting with the field name, a space, then the value, all in null-terminated ASCII text. Although the relevant data is of variable length, a fixed size block of 81 bytes is transmitted for every line, making it easier to separate the lines on the receiving end. Some fields are longer, having several subfields, each of which has its own line, and begins with a tab character. The first two fields (lines) of any Archive message convey the version of the message protocol, and the message type.

Following the header, any data is sent as the final part of the message. Because the object size is given earlier in the Data Info field, the recipient knows how much data to expect, so a data terminator is not needed.

Fundamental Operations

The system uses the following basic operations for all interactions among the components of the Archive Service. In our implementation, interactions are messages, and each operation has a simple request and response. Operations are combined to carry out user requests, and perform data management. The operations and the expected return message are explained below. A Failure message with error code is returned for unsuccessful requests.

Store Data: This request provides a stream of bytes and asks for it to be stored in the Archive. The request is sent to a Storage Server to support a user store request (request sent by Access Point) or to move or copy an object from one site to another (request sent by File Manager or Administrative Client). The name must be included in the message because this is kept with the data. After storing the data, the Storage Server returns an Acknowledge message containing the location string that can be used to retrieve the data in the future.

Retrieve Data: This request is used to extract data from a Storage Server. To guard against stale locations, or pathological reuse of local file names, the server first verifies that the name in the message matches the name attached to the stored file. Then the object corresponding to the location field is returned in a Return Data message.

Update Data: This request is used to replace the data already stored with a name. It causes the local copy of the data to be replaced with the new data included in the request. Because the local filename may be changed by the replacement, a new location string is returned in the Acknowledgment.

Delete Data: This request is sent to a Storage Server to delete the data associated with the indicated object. It merely means to delete the particular copy of the object stored locally. This is done either to delete the object entirely, or as an administrative action, if the particular copy is no longer needed.

The Storage Server validates that the name provided matches the one stored with the file. Then, if successful, an Acknowledgment message will be returned. It can be made by a client, in which case it represents a user request to eliminate the object entirely.

Get Name: This request asks a Name Generator to provide a globally-unique name. The Generator builds such a name and returns it in a Return Name message.

Add Name: When a new object is created, this message is sent by an Access Point to a Name Resolver. It causes an entry to be added to the name table, along with a name to location mapping. On success, an Acknowledgment is returned.

Lookup Name: This request is used to ask a Name Resolver to return a location for the name provided. The host originating the request is included in the message, so that the Resolver may use this to choose the most appropriate location to return. On success, a Return Location message is returned.

Delete Name: This request causes a name to be completely removed from the name table. It is sent by an Access Point to a Name Resolver at the completion of a user request to delete an object. On success, an Acknowledgment is returned.

Add Copy: This request may be sent to a Name Resolver to add an additional name to location mapping to the name table. It is sent by a File Manager to carry out an administrative action such as replicating or moving a file. This request is only valid if the name already exists in the table. On success, an Acknowledgment is returned.

Delete Copy: This request deletes one of the name to location mappings in the name table. It is only successful if other mappings exist for the given name. In this way, a Name Resolver is used to serialize deletions, ensuring that, if two File Managers independently decide to delete their local copy of an object, they cannot remove the last copy of the data. On success, an Acknowledgment is returned.

Record Access: This message is sent by an Access Point to notify the File Manager of the type of access request received. Possible accesses are retrieval of local or remote data, stores, and deletes. This information is used by the File Manager to compile its usage statistics, which are used to decide when to replicate or move files. On success, an Acknowledgment is returned.

Control: A Control message is currently intended to be an orderly shutdown request to a server. However, the File Manager uses the Control message for a different purpose and the shutdown is ineffective for the Access Point. When the File Manager receives a Control message, it toggles the automatic file replication on or off. This is useful both for measuring the performance without replication, and to allow manual replication instead. When a Control message is sent to the Access Point, it merely causes a shutdown of the child which is spawned to handle the request. A solution would be using a signal which the server can catch, then proceed with an orderly shutdown. No response is sent for a Control message.

Lock Name: This request is sent to a Name Resolver to mark a name so that other locks or change requests will be queued until the name is unlocked. On success, an Acknowledgment

is returned. *Not implemented.*

Unlock Name: This request is sent to a Name Resolver to unmark a name so that other change requests can proceed. On success, an Acknowledgment is returned. As with other features of the Service, Lock and Unlock are defined as separate requests for simplicity. As an optimization, they could be combined with other requests. *Not implemented.*

Lookup All: This request is sent to a Name Resolver to request all the locations for an object. On success, a Return All message with a list of locations is returned. This is used by an Access point to process a Delete or Update primitive, which requires all locations to be affected. *Not implemented.*

Replace Location: This request to a Name Resolver replaces the entire list of locations for an object. To preserve the access counts, any locations that directly replace existing copies must provide both the old and new location. On success, an Acknowledgment is returned. *Not implemented.*

Archive Management

This section describes the algorithms used to manage the data stored in the Archive. This includes making the decisions to add or delete extra copies of the data, and how the data is copied or removed using the fundamental operations.

As noted in Dowdy and Foster [17], an optimal file placement solution is not reasonable to expect, and even good heuristics can be tricky. Therefore, we adopted a simple algorithm, as our primary goal was implementing the method for file management, not the policy. The algorithm is executed by the File Manager at each site using local information only. The objective is to acquire local copies of data that is frequently requested at the site, and get rid of copies that are infrequently accessed. The following access counts are collected from the Access Point:

- Retrieve request for local data
- Retrieve request for non-local data
- Store request (always stored locally)
- Delete request. *Not implemented.*

For each object, we record the time of last access and an aged frequency count, which gives more weight to recent accesses by multiplying the old count by 3/4 each time the algorithm is run. This AGE_FACTOR is defined in a header file. The time between executions of the algorithm is configurable, but we envision once a day as a reasonable frequency.

The algorithm has two guidelines. The first is to keep the disk space utilization within a specified range, the second is to keep local copies for files that were most frequently accessed. To decide which files should be kept locally, the algorithm scans its list of files, adding and deleting local copies until these parameters are as close to their targets as possible. The File Manager at each site executes the algorithm as follows:

1. The File Manager first determines which files should be replicated locally. A file is deemed to be replicable if it is non-local and its count field is greater or equal to some threshold.

2. The File Manager looks at its current disk usage, its allowed disk usage, and computes the number of bytes needed to satisfy the requirements of step 1:
```
extra_bytes_needed =
   bytes_wanted - (allowed_disk_usage - current_disk_usage)
```

3. The File Manager attempts to free up `extra_bytes_needed` by erasing the local files that were the least frequently accessed (please note that due to our aging policy, files which were heavily accessed in the distant past will have their count eventually decrease to zero). Files can only be removed if they are local, if there exists at least another copy stored on some other sites, and if their count field is less or equal than a low threshold. Note that because of these requirements, the File Manager may not be able to free exactly `extra_bytes_needed`.

4. In phase four, the File Manager replicates files until it reaches the disk threshold or until all replicate requests have been satisfied. Each replicated file is then marked as local.

The steps for replicating an object are shown in Figure 3. When a File Manager determines that it needs a copy of an object, it asks a Name Resolver where to find a copy. It then connects directly to the Physical Storage server on a site holding a copy. It retrieves a copy of the data from here and stores it with the local Physical Storage server. Finally, it notifies the Name Resolver module of the additional location for the object.

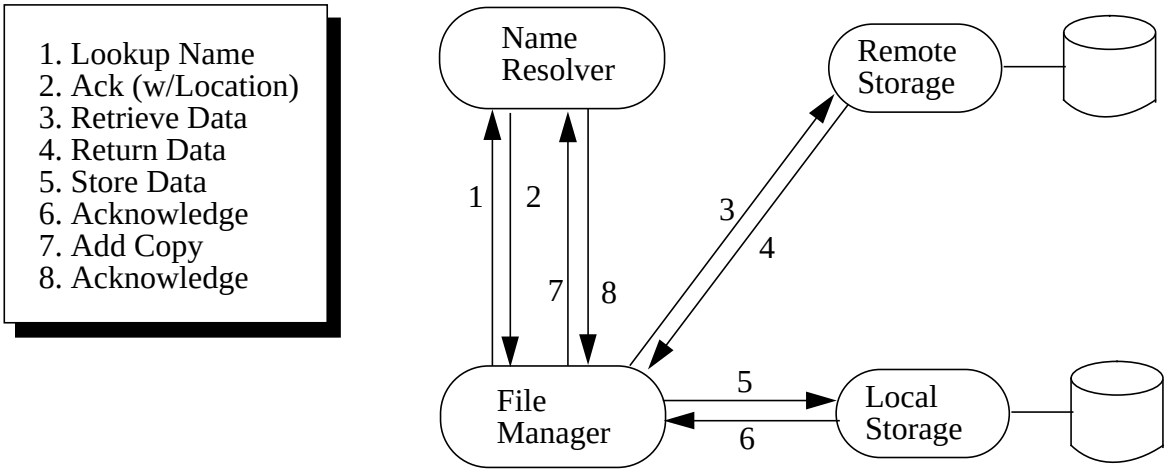

Figure 3: Replicating an Object

The steps for deleting the local copy of an object are shown in Figure 4. To ensure that the local copy is not the last one, the File Manager first deletes the location from the Name Resolver. This will succeed only if other copies exist. Then, the data is deleted from local storage to finish the operation.

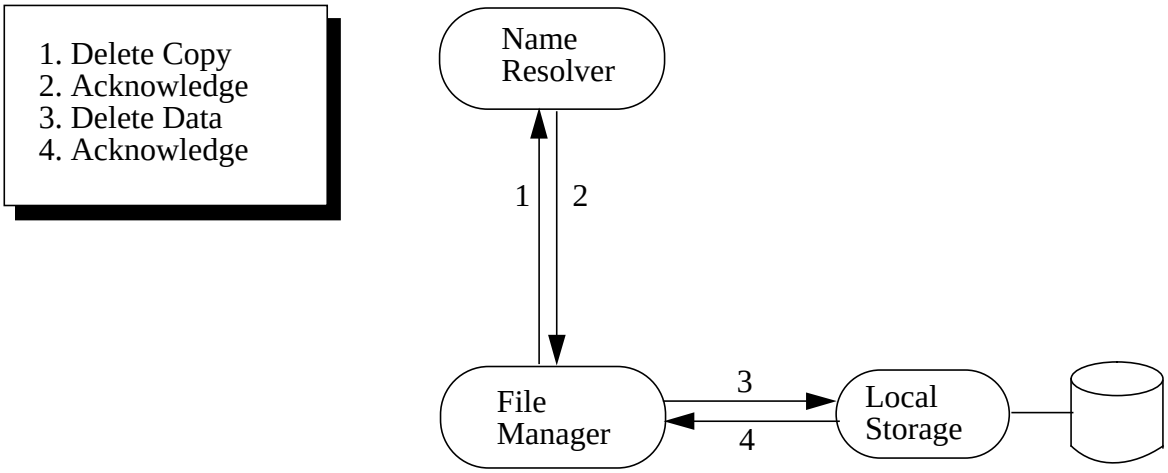

Figure 4: Removing a Local Copy

Note that an administrative client has not been implemented. This would be used to manually replicate files or delete extra copies. As a practical matter, this functionality could be incorporated into the existing command-line client.

Configuration

Each archive site has a local configuration file to allow setting parameters, debugging level, and identify the site which runs the name resolver. The configuration is read from the *archive.cfg* file in the current working directory when a process starts. If no file is found, default values are used, with the local host name used for the location of all the servers. The bulk testing clients also have their own configuration file (*client.ini*). Sample configuration files are shown in Appendix 2.

In order to make the system more scalable, there is no central configuration file which must be updated. Each site must have a pointer to a host with a Name Resolver, but otherwise, no configuration is needed to identify the hosts. A consequence of this is that there is no complete list of all sites in the Service. This restricts the algorithm for moving and replicating files to be done with local or "nearby" information only.

However, we view this as an advantage. A central file placement algorithm would not be able to execute quickly enough if the number of sites were to increase without bound. On the other hand, a site's file manager can decide how best to handle files considering local information plus information from the sites which share files with the site. If a distributed name resolver were introduced, then the file manager might normally only find out about a subset of the other sites which hold copies of a file, but this should be sufficient for a heuristic algorithm.

PRIMITIVES

This section describes the definition of each primitive provided to users and programs external to the Archive Service. This includes how it is constructed from the fundamental operations and the

data consistency model that is maintained when data is modified and retrieved simultaneously.

Store

This primitive creates a new object and binds a name to it. The user must save this name, and may pass it on to others. The object can only be retrieved in the future if this name is provided. Although the Archive Service provides no indexing or searching of documents, under "Supporting Other Naming Models" on page 28, we discuss how to add these features on top of the Service.

In submitting a Store request, the user provides a stream of bytes (generally, a filename is given to the client) and the desired access permissions. The Service stores the data, and returns a permanent name. This interaction, starting with the request, and ending when the name is returned to the client, is shown in Figure 5.

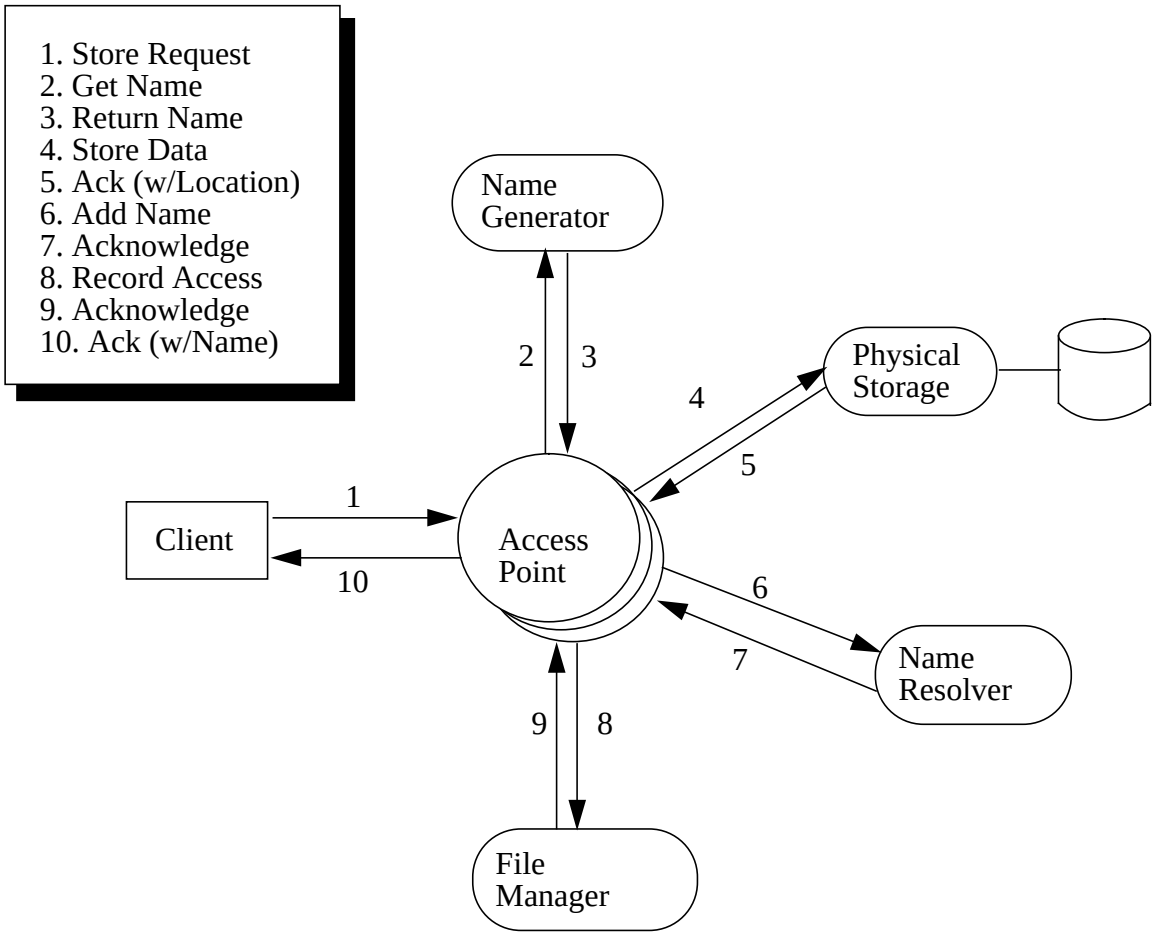

Figure 5: Execution of the Store Primitive

Once the name is returned, the object is guaranteed to exist and be accessible from every site in

the Archive Service, if the name is provided. It will actually be available slightly earlier, but this should not be a problem.

Retrieve

When the user provides a name in a Retrieve request, the Service returns the data stored under that name. If a location is included in the request, the client extracts the host name from the string and uses this as the site to connect to, rather than the usual Archive site. *Note that the current client does not implement recognizing a location string*. When the Access Point receives the request and sees that a local location is included, it will first try to retrieve a copy directly from the local Storage Server. If no location is included or the location is no longer valid, the Access Point goes to the Name Resolver to find the current location. If this is local, the Storage Server is contacted, and data is returned to the user. If the data is not local, the Access point will instead return a Referral message, and the client will contact the Access Point at the storage site to finally get the data. A Retrieve request, where the data is stored locally, is shown in Figure 6.

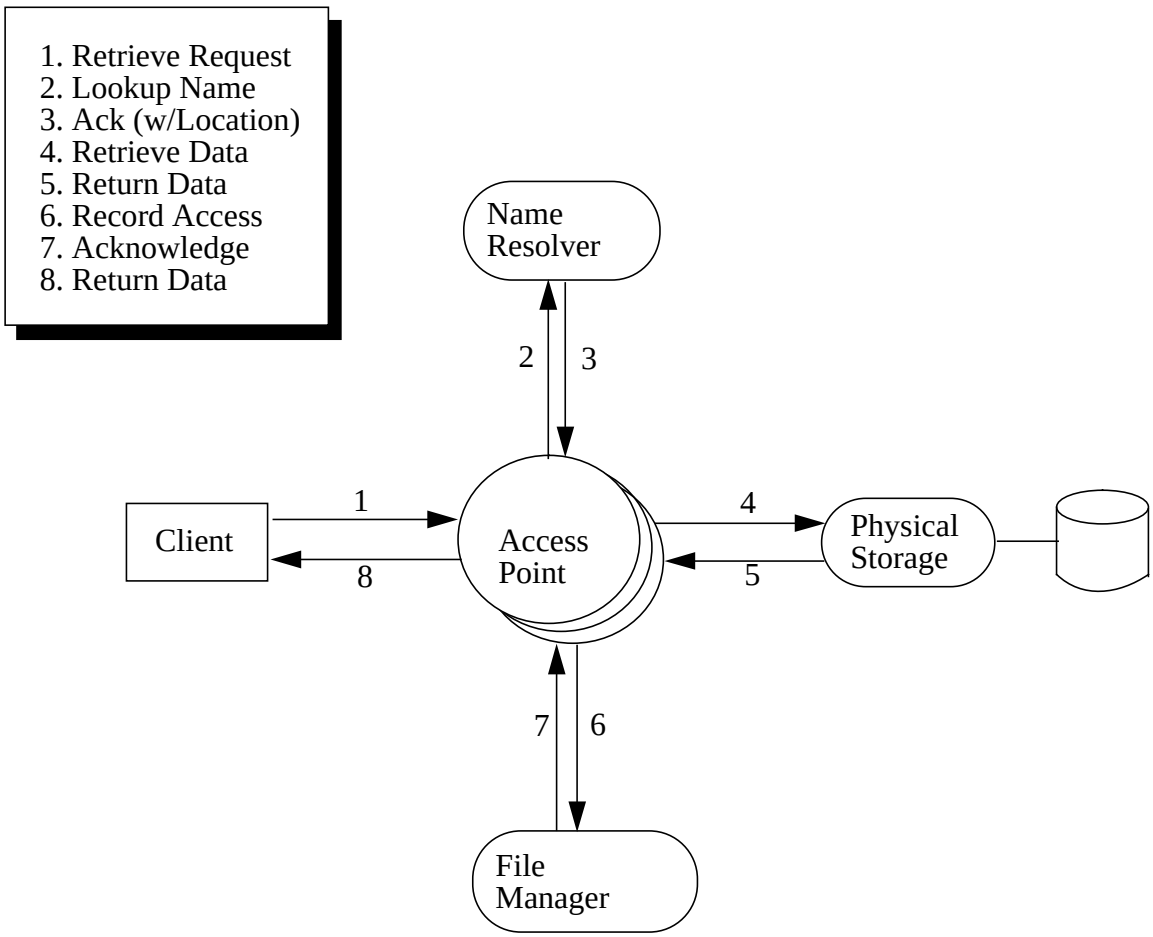

Figure 6: Execution of a Retrieve Primitive with Local Data

If the data is remote, step 8 becomes a Referral, and then the client sends a Retrieve request with a location string to the remote Access Point. This Access Point sends a Retrieve Data request to its local Storage Server, and the execution continues as in steps 4 to 8 in Figure 6.

If the original request contains a valid location, then steps 2 and 3 are skipped, and the request can be serviced without making a remote connection (beyond the Access Point) or using a central resource. However, if the request contains a stale location, the Storage Server will send a Failure in response to the Retrieve Data request. Then, the Access Point proceeds as if there were no location in the request, sending a Lookup Name message to the Name Resolver, etc.

Update

The user provides new data to store under an existing name. After verifying that the user is indeed the owner, and that the object is not immutable, the Service replaces all copies with the new data.

In the model we adopted for the semantics of the Update request, some inconsistency is allowed while the request is executing. Prior to the start of the request, all Retrieve requests will get the existing data. After the user provides a replacement file for update, some Retrieves will get the old data, and some will get the new, as the Update progresses. At some point, all sites will have the new data and all Retrieve requests will get this. Finally, the user will receive an Acknowledgment to the Update.

This model is easy to implement, and requires only a way to ensure that only one change is made to an object at time. An Update is executed as follows (also shown in Figure 7):

1. Go to the Name Resolver, lock the name, and get the full list of locations.
2. Contact each location, one at a time, and change the data, getting new physical names each time.
3. Go back to the Resolver. Provide the new names and unlock the object.
4. Send an Acknowledgment to the updating user.

Other than the Access Point and the File Manager, the processes may be on any host, not necessarily the site with the Access Point.

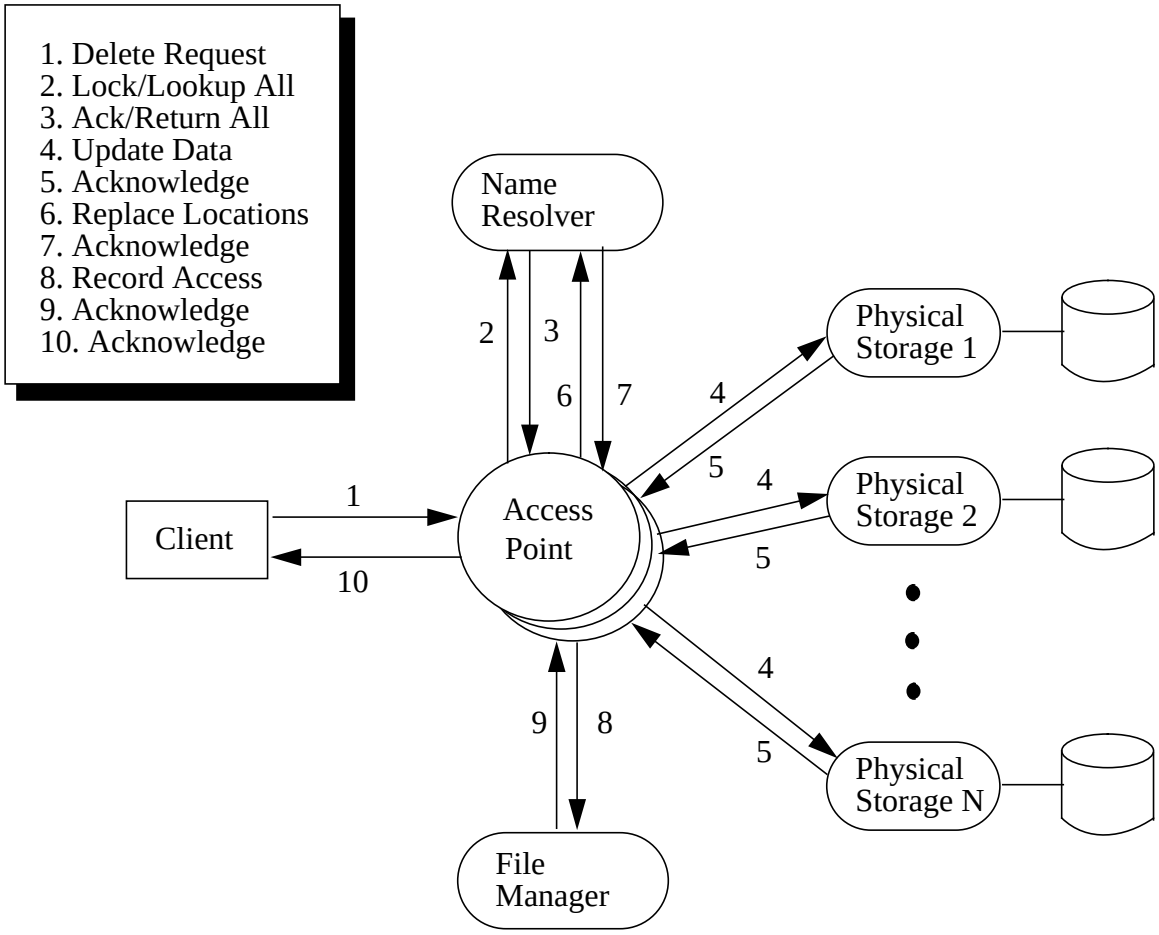

Figure 7: Execution of an Update Primitive

If a user does not receive an acknowledgment, it is either because the request is taking an unusually long time (due to high load or failed sites) but will eventually arrive, or else that the serving site has gone down.

Updating the copies one at a time is the simplest in the access point, but causes longer delays for the updating user, and for others who need to change this data. Especially because one or more sites may be down, causing timeouts, parallel execution is desirable. However, it is more complex, and care must be taken to avoid a solution that performs worse than the sequential one. For example, spawning child processes may involve making extra copies of the data, and incur process start-up costs. Lightweight threads should be feasible because each subthread can update the location mapping for its own copy, and the main thread can just wait on all the others to terminate before returning the Acknowledgment.

Delete

The user requests that a name (and attached data) be removed from the Archive. By the time the

user receives the Acknowledgment, the object and name will be deleted, and not accessible from any point in the Archive. Retrieve requests will continue to be serviced until the final copy is deleted. There is a possible race condition such that a client may receive a Referral for an object but find that it does not exist by the time it contacts the storing site. To shield the user from this anomaly, it is treated by the client in the same way as if the object never existed.

The following steps are performed for a Delete primitive:

1. Retrieve the full list of locations from the Name Resolver, placing a lock on the name.
2. Contact each location and delete the copy of the object stored there.
3. Delete the locations from the name table as the local copies are deleted.
4. Delete the name from the Name Resolver. This implicitly causes the lock to be removed, and all pending requests fail.
5. Send the acknowledgment to the user.

A Deletion is illustrated in Figure 8. Other than the Access Point and the File Manager, the processes may be on any host, not necessarily the site with the Access Point.

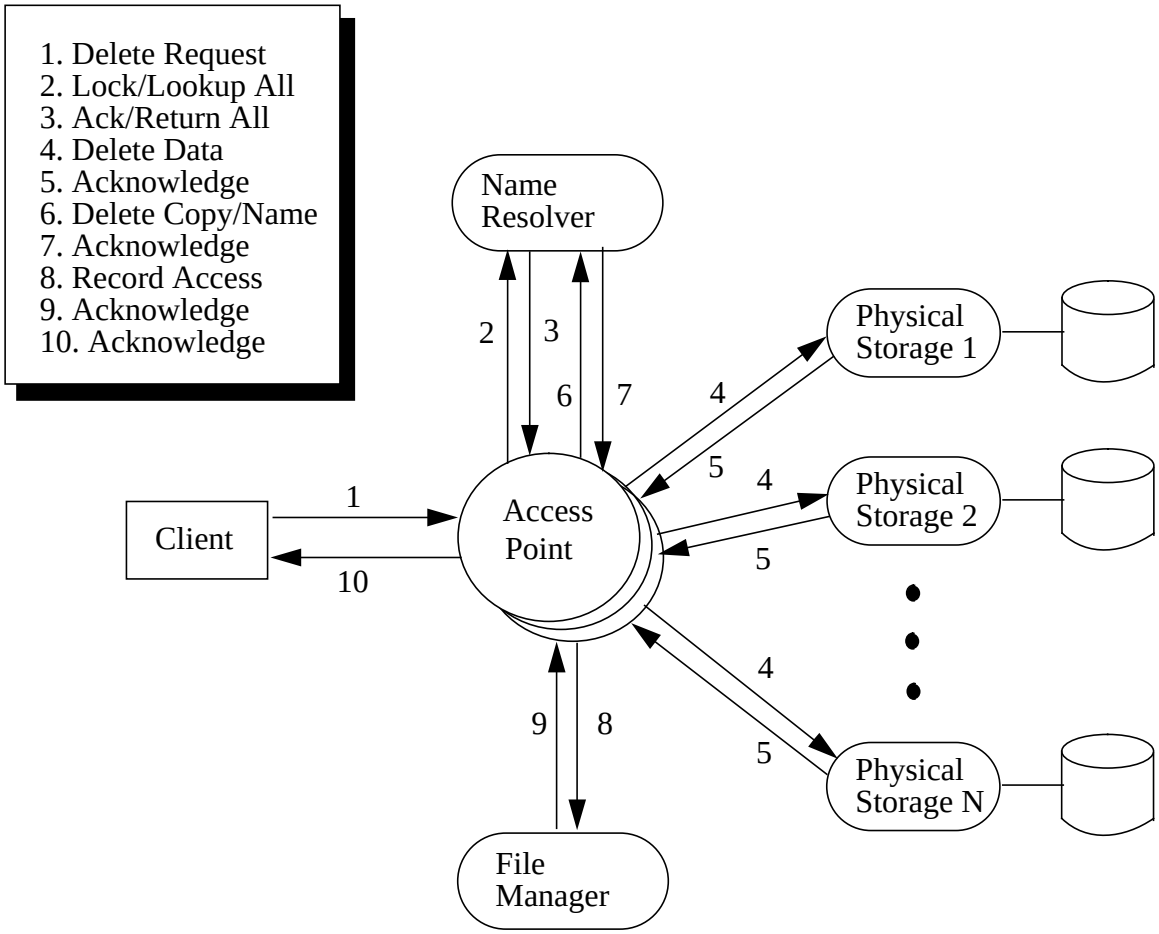

Figure 8: Execution of a Delete Primitive

As with the Update primitive, the storing sites are contacted one at a time. As with Update, it would be preferable to do this in parallel. Because each site does not need to be given new data, and because no new location must be returned, parallel execution would not be as complicated for Delete as for Update.

When it learns that the object is deleted, the File Manager can flush the name from its tables. File Managers at other sites may still be left with useless records for this name, but these will eventually be eliminated in one of the following ways. If the File Manager requests a local copy of the data, it must first obtain a location from a Name Resolver. It will then learn that the object no longer exists, and delete the record. If a client requests the object through that site, the Access Point will discover that it is gone and notify the File Manager, which in turn deletes the record. Finally, if the name is never requested at that site, its access count will be aged to zero and the record removed from the File Manager.

Describe

This primitive is a request to return the characteristics maintained by the Archive. These characteristics are listed above.

This request can be processed similarly to a Retrieve request. If the object is stored locally, the characteristics could, in principle, be retrieved from the Storage Server. *However, the Storage Server does not currently maintain this information.* Otherwise, a Name Resolver is contacted. However, as the characteristics are kept in the name table, it is not necessary to go to storage at this point. The execution of a typical Describe request is shown in Figure 9.

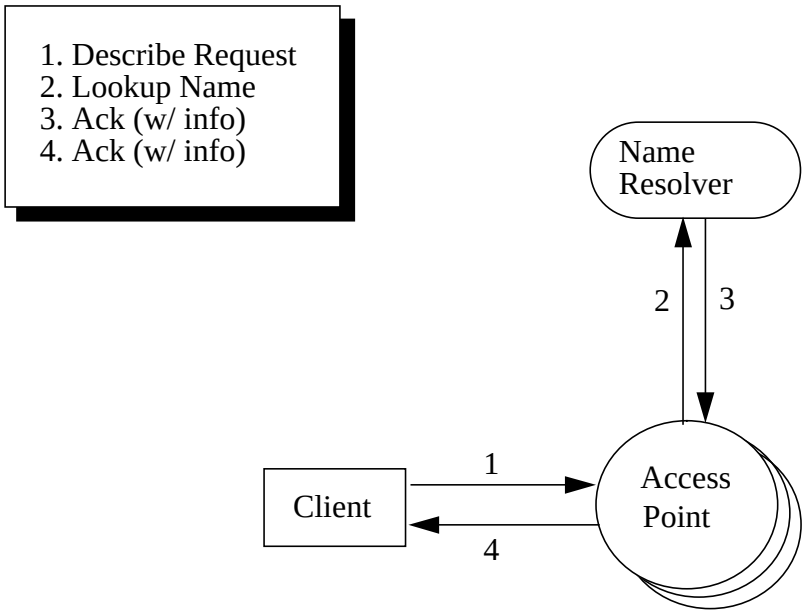

Figure 9: Execution of a Describe Primitive

MAINTAINING CONSISTENCY

We have identified several types of errors and race conditions that may occur. Below, we explain how consistency is maintained in these situations:

1. Two users request an update simultaneously
2. An update and a delete are requested simultaneously
3. File management tries to copy or delete a file during an update.
4. User disconnects while a request is proceeding.
5. Failed site: If there are several sites storing the data one of these may fail. If the data is only stored at a single site, it is also possible for this to fail, making it impossible for the request to proceed. Finally, the site that the user is connected to may fail.
6. Network Partition

These schemes all rely on the fact that our implementation uses a centralized Name Resolver. As this is not scalable, a discussion of how to apply this to a distributed resolver is included under “OPERATIONAL ISSUES” on page 29.

Simultaneous Changes

If any number of Update and Delete requests are made simultaneously for a single object, they will be serialized by the Name Resolver. Because all such requests must go through this point, only one will be allowed to proceed at a time, ensuring consistency.

Because File Manager and administrative requests also go through the Name Resolver, these operations can also be prevented from interfering with each other or user-initiated changes. *However, because these requests do not lock the objects they manipulate, they are not currently safe. If a File Manager retrieves a copy of an object, then a user replaces the data, and finally the File Manager stores the old data at a new site, the Archive will be inconsistent.* The obvious solution, thought not implemented, is to have these requests lock objects before manipulating them.

User Disconnect

If the user disconnects during a Retrieve or Describe request, the execution is aborted as soon as the disconnect is detected. This is done because the operation is seen as pointless if there is no user to receive the output. If the user disconnects during an Update or Delete request, the operation is completed anyway. This avoids the need to support an undo operation to abort a change when it has propagated to half the sites holding the data. This does leave the user in the dark as to the completion of the request, but the change can always be requested again, to make sure it is executed. (Note that retrieving the object to have a look at it would be a practical solution, but the Archive does not guarantee an upper bound on how long a change may take, so the user cannot be sure that a change did not occur just because a Retrieve request returns the old data.)

Failed Sites

To handle failed sites, we set a timer when connecting to a site. We assume a site is down if it fails to respond or a connection is not accepted within a timeout period. The timeout should be set long enough that there is a reasonable expectation that it will only expire it the site is truly down. There is, however, no guarantee that this will be true. Also, this does not protect against

network partitions, a problem that is discussed below. *Note that this timeout is not implemented. If a site is completely down or unreachable, the socket will eventually (after 60 seconds?) fail on a connection, and that will be recognized as a failure. However, the application itself currently sets no timeouts.*

To maintain consistency, a site would be responsible for applying missed updates when it comes up. To support this, a log of deletions and updates is needed for any site that is down. Retrieve requests to a down site will simply fail and do not need to be logged or executed later. *This level of robustness has not been implemented, and there is still an open question of how and where to keep the update/delete logs.*

It would also be possible to hold up processing a request until a site responds. If the site is down, the request would keep trying to connect until the site recovered and the request could be executed. However, if the site stays down more than a short while, this is desirable neither for queries (Retrieve and Describe) nor for changes (Delete and Update). For queries, the user may not wish to wait so long. Users could, of course, implement their own timeouts and abort the query.

Changes are different, however, because they lock out other change requests. A stalled change request would also hold up all other sites trying to change the data. Because only a subset of the storing sites would be updated, this would also cause the Service to provide an inconsistent view of the data for an arbitrarily long time. Furthermore, it becomes more likely for the coordinating site to fail during this time, leaving the Archive in an inconsistent state. (Note: to avoid this, the site could -- and should -- keep a log of updates in progress. Upon recovery, it could re-execute any pending requests.)

Network Partition

We have to accept that we cannot tell the difference between a failed site and one that is separated from us by a network partition. Further, if the routers are not completely effective at finding alternate routes, there may be network failures that disable communication between two points without partitioning the sites into disjoint islands. Because of these problems, there may be a breakdown of consistency if a partition occurs.

However, this will be limited to a temporary inconsistency for queries. Sites that receive requests for locally stored data may serve them using old data, even if the object was updated in another part of the network. But, when the network recovers, the same procedure used to restore consistency to failed sites would work here. So long as the network failure neatly divides the sites into disjoint islands, eventual consistency is assured because Updates will only proceed in the island containing the Name Resolver. If, however, two sites can both reach the Name Resolver but not each other, then the Archive may become inconsistent. This is an unresolved weakness in the system.

INTERACTING WITH USERS

Unix Client

The most direct way a user can interact with the Archive Service is by using the command line client. It allows users to store and retrieve objects or send a control message to any of the Archive

servers. Although this should be an option on the command line, the client currently reads the server host from a configuration file (if present) in the working directory (default is local host).

The client is named "pac" for "Persistent Archive Client." The interface is simple, the name of the client, the requested operation in lower case, and any necessary arguments. The possibilities are shown below.

```
pac get <name> [<location>]
pac put <filename>
pac control { <port> | ap | phys | fmgr | res | gen | all }
```

For "get", the arguments are the name of the object to retrieve, and an optional best-guess location string. The client uses these to build a Retrieve request. If a referral is received as the response, the client attempts to get the data from the new site before returning to the user.

For "put", the only argument is a filename to read data from. The client currently creates only objects with no delete or update permission, although options should be added to override these defaults and/or change the defaults to allow changes. The data is read from <filename> to build the Store request.

For "control", the argument is the specific server to send the Control message to. This may be one of the mnemonics shown for the Access Point, Physical Storage, etc., or the integer TCP port may be given.

Gateway to the World Wide Web

We set up a gateway to demonstrate how useful a location-independent name would be in the World Wide Web, and to show how our approach could be used. As the first interface for Web clients, we ran the hypertext server (httpd) developed by the National Center for Supercomputer Applications (NCSA).

To use the gateway, a Web client opens a URL similar to

```
http://gateway-host:port/ARCHIVE/formget?190.3.0.0
```

This causes httpd to call a shell script using the Common Gateway Interface (CGI) [18]. The script serves as a gateway, parsing the logical name out of the parameters from the hypertext server, and uses this to invoke a retrieve request with the Archive Client. The client's output is returned to the Web browser via the hypertext daemon. This arrangement is shown in Figure 10.

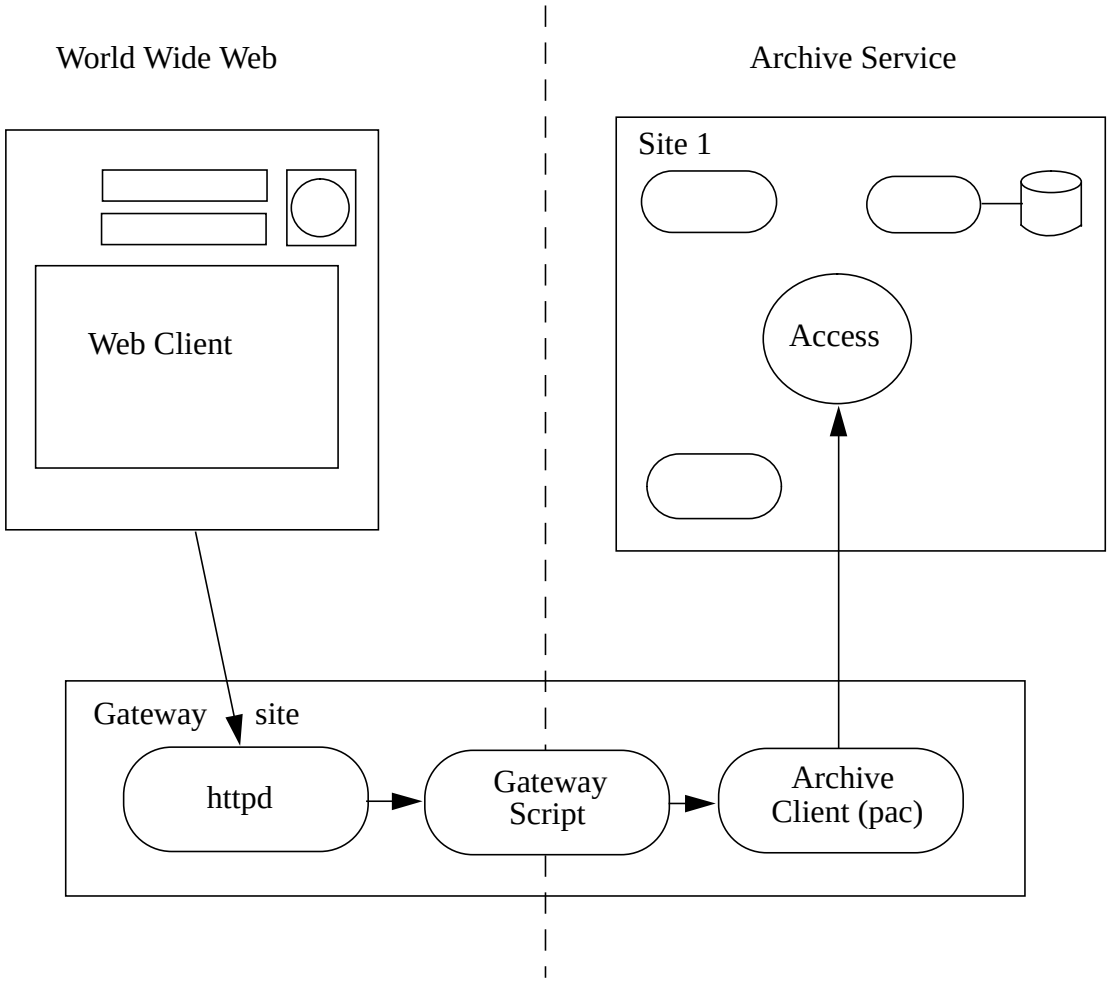

Figure 10: Gateway to the World Wide Web

To properly integrate location-independent names, it will be necessary to modify a WWW client to accept an Archive name. The client would be free to locate an Archive site whenever necessary, allowing a new site to be found if the current one becomes unavailable. The advantage of this approach is that it gets any location-dependence out of the name string. Because we did not modify a client, our demonstration was forced to use a URL containing the address of a gateway to the Archive. Although the address for any gateway could be used, it must be chosen when the URL is constructed, leaving us with names that are still location-dependent and prone to becoming invalid when a server is removed.

Finding an Archive Site

We suggest using the Domain Name Service (DNS) to choose an initial Archive site for clients to contact.[21][22] Clients could then save this site name in a local resource file only use DNS again if the site became unavailable. The demonstration we built did not use DNS for this purpose -- partly because we used unmodified WWW clients.

To use DNS, it is desirable to maintain a list of sites, from which one can be chosen as needed. By choosing well, the accesses from clients can be spread out across multiple Archive sites. The simplest method for maintaining a list in DNS is to have a pseudo hostname which resolves to a list of network addresses of actual hosts running Archive servers. As this is not the intended use of the many-addresses-to-one-hostname mapping, a more appropriate mechanism is desirable.

The canonical name (CNAME) and pointer (PTR) records seem to better fit our intended use, but their mappings are required to be unique. As no existing record type appears appropriate, a new one may be considered to replace the list of addresses described above. This may take the form of a service site (e.g. SERV) record type, including a service and the name of a host which provides the service. For flexibility, a port number could be added as well. To make a list, any number of service records may be given for a single names.

Once DNS is populated with a list of Archive sites, there is still the question of how to choose one of the sites when queried by a client. The way DNS works, the entire list will generally be returned to the client, so this choice can be made by the client. (If the list becomes too long, it may be possible to divide it hierarchically.) The best criteria for the client to use is not known. One possibility is to try to make the nearest match with the client's network address. Although this is a crude method it may be able to distribute the client accesses to some extent, and at least yield an Archive site on the same continent -- for most clients.

Supporting Other Naming Models

There are other models that need to be supported by a global archive service. As discussed under the related work on URNs by the IETF, users may wish to have names with a variety of semantics. The basic names in the Archive Service have the fixed meaning of pointing to a specific object, even a fixed set of bytes if the object is immutable. We wish to be able to provide a name for the following concepts:

- the latest version of an object.
- the intellectual content of a document (which may be available in several formats).
- a collection of objects.

We also feel it is necessary for users to have access to more detailed and descriptive attributes of an object than those maintained by the Archive Service. These characteristics might include title, author, format, possibly data compression, information type (text, image, video), a description, abstract, etc. This information could be viewed by users to decide which objects to retrieve, or used automatically by client to perform the same function, based on user preferences.

However, as noted previously (in "OUR APPROACH" on page 2), we believe that all these meanings can be supported without being part of the basic implementation for the service. We recommend adding an extra layer on top of the Archive Service. All requests to store, retrieve, find, delete, or modify an object would go through this add-on service, which can use the standard client interface to the Archive. As with the basic service itself, we describe the implementation as a separate process to clearly demonstrate the division of functionality. While this is a feasible approach, there is still no reason why all the functions could not be integrated into a single program to achieve higher performance.

To support a name for the latest version of an object, an added layer could use a mutable object (also stored in the Archive) to hold the name of the most recent item in the series. Note that this item may be immutable. To provide the "latest version" service, this layer would give out the name of this mutable object. When a request comes for this name, the layer retrieves the object attached to it. Looking inside, it sees that this is just a pointer, extracts the name it points to, retrieves the actual data, and returns the data to the client.

To make this work, the added layer can store a header in the beginning of every file it stores in the Archive. When a file is retrieved, the layer checks this information first, to determine whether to return the data to the client, retrieve another object, etc. It would also be possible for the service to maintain its own lists of pointers outside the Archive, however, this does not seem to be necessary, and using the Archive builds upon the service it already provides.

A name for a collection of objects would, in the scheme described, be a name for a pointer object. This is a small object containing the names of each individual object in the collection. The object could also contain other information, such as data about the collection, etc. When the add-on service retrieved such an object, it would see in the header that this is a pointer object, and retrieve all the objects in the collection, and return these to the user.

Because of the large and variable amount of data they may contain, the detailed attributes listed above should be stored in a separate file, rather than maintained in the name table. This information for an object could be stored with either of the following approaches:

a. The attributes are included as tag/value pairs within the header, distinct from the data part. This approach means that the entire object must be retrieved, even when only the attributes are desired. (Although the data part must only be sent as far as to the add-on service.)
b. Always use a "pointer" name that is a small file containing the attributes and the name that holds the actual data. This approach requires two retrievals per object, if the characteristics are not requested first.

Either of the above schemes would be reasonable if the services were fully integrated onto a single process. However, the two-object solution would likely be preferable, as it avoids disk accesses when the data is not needed.

As an alternative solution to the one described above, the add-on service could use a hypertext format with a way to insert hidden tags to hold attributes, etc. These tags would not be part of the data returned to the client, although they might be put in a message header. Using hypertext would especially make it easier to handle collections of items. They could be included as inlined images, hypertext menus, etc.

Note that we do not describe the following service:

- Locate a document using an ad-hoc query on certain keywords or attributes such as author and title.

While indices can be made for a limited document collection, we feel it is not feasible to execute such a query in an unrestricted wide-area system which may contain billions (or more) of data objects. While one may attempt to construct an author or title index, searches would not necessarily be constrained to these attributes, so a search would in general require retrieving information

from every object in the archive. Before this is possible, there need to be effective approaches for identifying subsets of the Archive and for restricting searches to a small number of these sets.

OPERATIONAL ISSUES

Performance

Our intention was to develop an efficient and scalable architecture for this Archive Service. However, there are two major factors limiting the performance of the current implementation. These are the centralized Name Resolver module and the overuse of inter-process communication. The communication is made worse because the message passing model (a new TCP socket is opened for each message) is inefficient. All other parts of the system are distributed, work primarily with local information, and never require contacting all other sites.

To replace the Name Resolver with a scalable implementation, the name space should be divided up into subsets. The resolver would be distributed across many sites, with a set assigned to each site. The resolver is designated the *home resolver* for each name in its set. It is authoritative for these names, and performs locking and authoritative resolution. Dividing up the name space in this way allows the name resolution to be a scalable part of the Service. Because all change requests to a single object must go through its home resolver, consistency is maintained in the same way as with a centralized resolver. Implementing an efficient distributed name service is also discussed by Triantifillou and Bauer[19] and Cheriton and Mann[20].

To improve the communication efficiency, the first approach would be to combine all the modules at a site into a single server process, as has been noted previously. Another step would be to try and avoid the impact of TCP's connection model. All the interactions in the Archive Service follow a transaction model, with a simple request and response. By contrast, TCP requires three messages to be exchanged, just to set up a connection. A communication service supporting the transaction model would help. We might try to avoid the problem by maintaining permanent TCP connections, but because of the large number of sites potentially involved, this would not be feasible. However, once the modules on a single site were integrated, remote connections would be less of a problem -- needed mainly to call on a Name Resolver or replicate files (and for clients).

Robustness

In a realistic system, more attention must be given to fault tolerance (specifically, guaranteeing the consistency of the service in the presence of faults), high service availability (achieved mainly through redundancy), and administrative tools. We would want administrative tools to monitor the system, manually move or replicate files, even move all the files off of a site and shut it down.

To improve availability and fault tolerance, the Service would require redundant resolvers. This makes it more complicated, but not impossible, to ensure consistency. We would recommend using 3 home resolvers per set. In this way, two resolvers can get together and vote the other one dead. To make this work, the resolvers must continually send keep-alive messages to each other. However, because of the small number (2) of home resolvers per set, they can easily keep track of each other, making this reasonable. Because the name space is divided as described above, this would still be a scalable solution.

The main improvements to fault tolerance needed are for all servers to store their internal tables on disk, and keep logs of Updates and Deletes. Strictly speaking, the tables must be backed to disk whenever they are modified. Update and deletion logs are discussed above under "Failed Sites" on page 24, and are subject to the usual requirements (such as checkpointing) that apply to distributed databases. Also, as noted there, a failed site must be prepared to spend some time on recovery to bring itself up to date with the rest of the Service.

Security

Both within and between sites, a method is needed to authenticate connections. Other than at the Access Point, all accesses that do not come from other Archive Service processes must be kept out. Authentication is also needed for users accessing the Service. The name of the user that stores an object is currently stored with an object, and the Access Point should ensure only that user (the "owner") is allowed to modify it. However, to enforce this, we must be able to reliably identify the user, which we do not currently know how to do.

It may also be desirable to allow read access to an object only to some group of users. Because the Service must be scalable, using access control lists for each object does not seem feasible. Another approach is needed, possibly using capabilities.

TESTING

Tools

Flexible Client: To initially test the software, we wrote a client that sends a pre-built Archive message to a server. This client had a rather fixed interface, but could be used to test any of the processes in the system, merely by using that server's input port as a command line parameter. The messages were stored in files, which we made with a text editor, and redirected into the client's standard input. If the message included a data part, the file containing the data was given as the last line of input to the client. By not requiring the data to be in the text file, the client is able to handle binary files. Finally, the client waits for a response, and writes the entire message to the standard output.

Bulk Clients: We wrote two clients to use in performance tests. The feature of these clients is that they could generate a large number of requests. One, `store`, would create a number of objects using a uniform random distribution for the file size, and location. The other, `retrieve`, would retrieve those files, picking one at random and using an exponentially distributed inter-request time.

The storer and retriever communicate via a file (`lnames.all`) containing the names of all the objects created by the storer. This file is also read by the analysis tool.

Analyzer: A final tool takes an output file generated by the `retrieve` bulk client, and generates data suitable to make a histogram showing the time per byte to access data from the Archive, versus time. The output from the retriever lists each object, its size, the time it was requested, and when all the data was received. The `analyzer` then divides the accesses into discrete time periods of configurable length. All accesses within a time period are averaged, weighted by the number of bytes retrieved, to generate a time per

byte for the period. This method of combining is not necessarily the only or best one, but the tool could be modified to summarize the data in a different way.

Environment

We used four Sun 4c machines (Sparc IPCs) running SunOs 4.1.2. One host was chosen for the central Name Resolver. The directory `/tmp/archive` on the local disk of each machine was our physical store and the working directory for the server processes. We configured the sites to limit the Archive to 2.5 MB of data per site, to avoid interfering with other users.

Methodology

Other than verifying that the basic operations of store and retrieve work correctly, we ran some tests to see how the system performed under load. These tests also generated enough traffic and data objects to see the automatic replication in action.

The first test ensures that the system can handle a large number of retrievals, performed one after another. By running storers on multiple hosts, we could test simultaneous requests, and increase the load as much as desired. The size of the files and the sites they are stored on is chosen at random by the storer. We felt that a uniform distribution for these sufficiently modelled real-world scenarios.

For the next test, we used the retriever to access the files from the system. the retriever queries its local Access Point for the files by supplying only the logical name. The AP responds either by giving the file or by giving a “referral” message back. `retrieve` uses this referral message to find out an actual physical location of the file and then establishes a connection with the AP of that site to get the file. Between requests, we introduced a delay parameter. After making a retrieval request for a file and procuring it, a retriever goes off to sleep for a random time before the next request. The delay is an exponentially distributed variable because we feel that this models real-world scenarios more effectively.

We used the retriever to access files in the system a number of times and measured performance by noting the time it took to transfer the bytes. Next, we ran multiple retrievers over different machines to test the stability of the system. We found that having more retrievers is definitely a performance hit, but the protocol itself does not fail.

We also ran the setup once where the storer knew only about a single site. This defeated its random file placement, and caused all files to be concentrated at that site. We then ran retrievers on all machines. Initially, all Retrieve requests (other than at the main site) were serviced by referrals. Then, after the File Manager ran the replication algorithm, retrievals were handled locally.

WHERE THE CODE IS

The code and documents are stored under the directory:

```
~cyberia/projects/jjones.
```

There are five (somewhat arbitrary) directories for the code:

`include` -- contains the common header files.
`generic` -- contains the common libraries and the clients (cli and pac).
`access` -- contains the Access Point.
`code` -- contains the Name Generator, Name Resolver, and File Manager.
`storage` -- contains Physical Storage, bulk clients (store and retrieve), and the analyzer.

In addition, there is a `bin` directory with the start-up scripts and links to the executables. The documents are in the `docs` directory.

More details, and start-up instructions are included in a `README` file.

CONCLUSION

We have described the implementation of a distributed Archive Service. The Service is meant to be global in scale and provide a guarantee of name persistence. Although we did not implement a complete, deployable service, we believe that we have demonstrated a viable approach for providing this, and described how the missing pieces could be built.

ACKNOWLEDGEMENT

The initial version of this project was developed with Anand Natrajan, and Anh Nguyen-Tuong. At that time, the majority of the code was written. Minor portions of the original report have also been included, modified or unmodified, in this document.

REFERENCES

1. Berners-Lee, T., "Universal Resource Identifiers: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World Wide Web," *RFC 1630*, June 1994.

2. Berners-Lee, T., "Uniform Resource Locators (URL)," *URI Working Group Internet Draft: draft-ietf-uri-url-03.{ps,txt}*, <http://info.cern.ch/hypertext/WWW/Addressing/URL/url-spec.txt>, March 21, 1994.

3. Sollins, K., Masinter, L., "Requirements for Uniform Resource Names," *URI Working Group Internet Draft: draft-sollins-urn-01.txt*, June 30, 1994.

4. Mealling, M., "Encoding and Use of Uniform Resource Characteristics," *URI Working Group Internet Draft: draft-ietf-mealling-urc-spec-00.txt*, July 8, 1994.

5. "URN to URC resolution Scenario," *URI Working Group Internet Draft: draft-ietf-uri-urn2urc-00.txt*, February 19, 1994.

6. Mealling, M., "Specification of Uniform Resource Characteristics," *URI Working Group Internet Draft: draft-ietf-uri-urc-00.txt*, April 5, 1994.

7. Archive of the IETF URI Working Group Mailing List, <http://www.acl.lanl.gov/URI/archive/archives.html>.

8. Sollins, K., "URN and citations," *Message on URI Mailing List*, <http://www.acl.lanl.gov/URI/archive/uri-94q2.messages/58.html>, April 14, 1994.

9. Moore, K., "Re: URN and Citations," *Message on URI Mailing List*, <http://www.acl.lanl.gov/URI/archive/uri-94q2.messages/74.html>, April 16, 1994.

10. Gargano, J. and Weiss, K., "Whois and Network Information Lookup Service Whois++", *WNILS Working Group Internet Draft: draft-ietf-wnils-whois-lookup-01.txt*, June 26, 1994.

11. Deutsch, Schoultz, Faltstrom, Weider, "Architecture of the WHOIS++ Service", *Network Working Group Internet Draft: draft-ietf-wnils-whois-arch-00.txt*, April 6, 1994.

12. Huitema, Pays, Zahm, Woermann, "Simple Object Look-up Protocol (SOLO)", *Network Working Group Internet Draft: draft-huitema-solo-00.txt*, December 1993.

13. Touvet, J-C, Pays, P-A, "Directory Services and WWW integration using SOLO," *Inria*, <http://champagne.inria.fr/People/JCT/SOLOWWW/1.0.ps>, Version 1.0.

14. Kahle, B., "Document Identifiers or International Standard Book Numbers for the Electronic Age," *Thinking Machines Corporation*, version 2.2, September 1991.

15. Rao, H. C., and Peterson, L. L., "Accessing Files in an Internet: The Jade File System", *IEEE Trans. on Software Eng.*, vol 19, no. 6, pp. 613-24, June 1993.

16. Gifford, Needham, Schroeder, "The Cedar File System," *Communications of the ACM*, vol 31, no. 3, pp. 288-298, March 1988.

17. Satyanarayanan, M., "A Survey of Distributed File Systems," *Annual Review of Comp. Sci.*, no. 4, pp. 73-104, 1990.

18. Levy, E. and Silberschatz, A., "Distributed File Systems: Concepts and Examples," *ACM Computing Surveys*, vol 22, no. 4, December 1990.

19. Dowdy, L. W. and Foster, D. V., "Comparative Models of the File Assignment Problem," *Computing Surveys*, vol. 14, No. 2, pp. 287-313, June 1982.

20. McCool, R., "The Common Gateway Interface," *NCSA*, <http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>.

21. Mockapetris, P., "Domain Names - Concepts and Facilities", *RFC 1034*, November 1987.

22. Mockapetris, P., "Domain Names - Implementation and Specification," *RFC 1035*, November 1987.

23. Triantifillou, P. and Bauer, M., "Distributed Name Management in Internet Systems: A Study of Design and Performance Issues," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 357-68, 1990.

24. Cheriton, D. R. and Mann, T. P., "Decentralizing a Global Naming Service for Performance and Fault Tolerance," *ACM Trans. on Computer Systems*, vol. 7, no. 2, pp., 147-83, May 1989.

Appendix 1: Message Types and Fields

```
Message type: control ( 1) has  2 lines.
Fields:
           version
           type


Message type: store ( 2) has 10 lines.
Fields:
           version
           type
           data info
           user info
           data


Message type: retrieve ( 3) has  7 lines.
Fields:
           version
           type
           name
           physical name
           user info


Message type: delete ( 4) has  7 lines.
Fields:
           version
           type
           name
           physical name
           user info


Message type: update ( 5) has 12 lines.
Fields:
           version
           type
           name
           physical name
           data info
           user info
           data
```

```
Message type: get_name ( 6) has  2 lines.
Fields:
           version
           type


Message type: store_name ( 7) has 11 lines.
Fields:
           version
           type
           name
           physical name
           data info
           user info


Message type: new_copy ( 8) has  4 lines.
Fields:
           version
           type
           name
           physical name


Message type: lookup_name ( 9) has  7 lines.
Fields:
           version
           type
           access point
           name
           user info


Message type: delete_copy (10) has  4 lines.
Fields:
           version
           type
           name
           physical name


Message type: store_data (11) has 11 lines.
Fields:
           version
           type
```

```
            name
            data info
            user info
            data


   Message type: record_access (12) has  9 lines.
   Fields:
            version
            type
            request type
            name
            physical name
            data info


   Message type: retrieve_again (13) has  7 lines.
   Fields:
            version
            type
            name
            physical name
            user info


   Message type: ack (15) has  9 lines.
   Fields:
            version
            type
            request type
            name
            physical name
            data info


   Message type: referral (16) has  4 lines.
   Fields:
            version
            type
            name
            physical name


   Message type: return_data (17) has  9 lines.
   Fields:
            version
```

```
            type
            name
            physical name
            data info
            data


   Message type: return_name (18) has  3 lines.
   Fields:
            version
            type
            name


   Message type: return_loc (19) has 11 lines.
   Fields:
            version
            type
            name
            physical name
            data info
            user info


   Message type: failure (20) has  5 lines.
   Fields:
            version
            type
            request type
            reason
            text
```

Appendix 2: Sample Configuration Files

Sample archive.cfg with storage directory in /tmp/archive:

```
debug   4
frequent 1
seldom 0
disk_quota 2500000
disk_threshold 2000000
directory   /tmp/archive/
generator_id    190
filealg_interval 3600
algorithm_set 1
resolver jade.cs.virginia.edu
```

Sample client.ini:

```
Debug 0
MaxFileLen 200000
MaxDelay 0
MaxFiles 10
AverageTime 500
Retrievals 20
TimeInterval 2000
RandomSeed 60376
AccessPoint jade.cs.virginia.edu
```