CITI Technical Report 93–11

Managing Heterogeneous Distributed Computing Systems: Using Information Repositories

Gerald A. Winters
gerald@engin.umich.edu

Toby J. Teorey
teorey@citi.umich.edu

ABSTRACT

An integral part of managing heterogeneous distributed computing systems is an information repository. The ultimate goal of our research is to specify a methodology for the design, analysis, and comparison of information repositories for such systems. We first outline the general characteristics of data repositories, including requirements and data model features. Then we build an experimental prototype system to test two candidate repositories: X.500 and the AFS file system. Performance and scalability measurements are collected, analyzed, and compared for the two platforms; and some preliminary conclusions are reached.

December 20, 1993

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Managing Heterogeneous Distributed Computing Systems: Using Information Repositories

Gerald A. Winters
Toby J. Teorey

December 20, 1993

1. Introduction

Managing heterogeneous distributed computing systems is an area of intense research. Loss of operations, or down time, in such a system represents a loss of resources, as well as dollars and cents. Thus, it is not surprising to see activity in standards, products, and architectures devoted to the management of heterogeneous distributed computing systems. An integral part of such a management system is the information repository. The goal of the repository is to store the information necessary to support successful management. The data includes dynamic information coming from sensor agents around the system, as well as static information about the system configuration. The repository thus provides information about the general state of the entire computing system. Figure 1 depicts a generic management architecture.

Note that the repository in Figure 1 is physically distributed. The figure represents a logical view, which to the user appears as a single entity. Managed entities can be hardware network devices, such as routers or file servers, or components of operating systems, such as mailers and process schedulers. Typically a software module, called an agent, resides in the entity being managed and stores status information about the managed object.

A management server, through the agent, can read and alter management information associated with a managed entity. The communication protocol between a management server and agent is usually based on a standard such as SNMP [4] (simple network management protocol) or CMIP [1] (common management information protocol). Figure 2 illustrates this relationship.

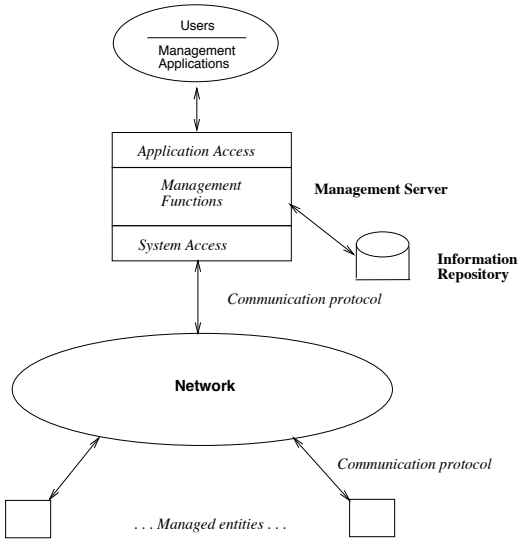

Figure 1. Logical view of a generic management architecture.

Figure 2. Manager/agent relationship

Many interesting research questions arise when specifying the repository: What kinds of data must be gathered to monitor distributed applications? Is it necessary to collect historical data? How should data be modeled? Relational? Object-oriented? Temporal? How will users access data? How should the repository be organized? What minimum performance requirements must the repository meet? Obviously, these questions are not entirely independent. The goal of this paper is not to give a definitive answer to each of the aforementioned issues, but rather to describe research and propose future directions.

The paper is organized as follows. Section 2 introduces CORDS, the research project we are part of. Section 3 describes the data, data model, and requirements of a generic information repository. Section 4 introduces an experimental prototype management system and discusses measurements made on two different data repositories. The paper concludes by outlining the directions for future research.

2. CORDS Management Architecture

The CORDS (Consortium on Research in Distributed Systems) project [14] is an IBM funded research project aimed at providing an environment for the development of distributed applications. Our research is to provide technology for CORDS and this section presents the CORDS architecture and gives a brief explanation.

The integrated management architecture shown in Figure 3 and proposed in [3] consists of three layers, application layer, systems layer, and network layer.

The application layer consists of the CORDS application tools, the tools available to the user for distributed applications development.

- NetMod [2] (Network Modeler) is a network performance tool designed to model arbitrary LAN networks.
- NEST [7] is a graphical-based environment for the simulation of distributed networked systems.
- NETMATE [6] is a comprehensive network management package.
- Hy+ [5] is a visual database system for managing and controlling large heterogeneous networks.
- Shoshin Event Monitor [15] supports debugging of distributed and parallel applications.

The systems layer consists of various services or subsystems, including the data repository, that carry out actions on behalf of the CORDS application tools. A brief summary of the main subsystems found in this layer follows:

- **Management Information Repository Subsystem.** The collection of information repositories for management information. This subsystem will contain information on objects as defined by SNMP or CMIP, for example, as well as information such as server availability and CPU load. This information is available to the administrator or manager and to the CORDS tools.
- **Configuration Subsystem.** This subsystem is responsible for the addition, deletion, or modification of managed objects and their agents that will be stored as part of the information repository.
- **Monitoring Subsystem.** This group of components is responsible for the monitoring of managed objects. This includes

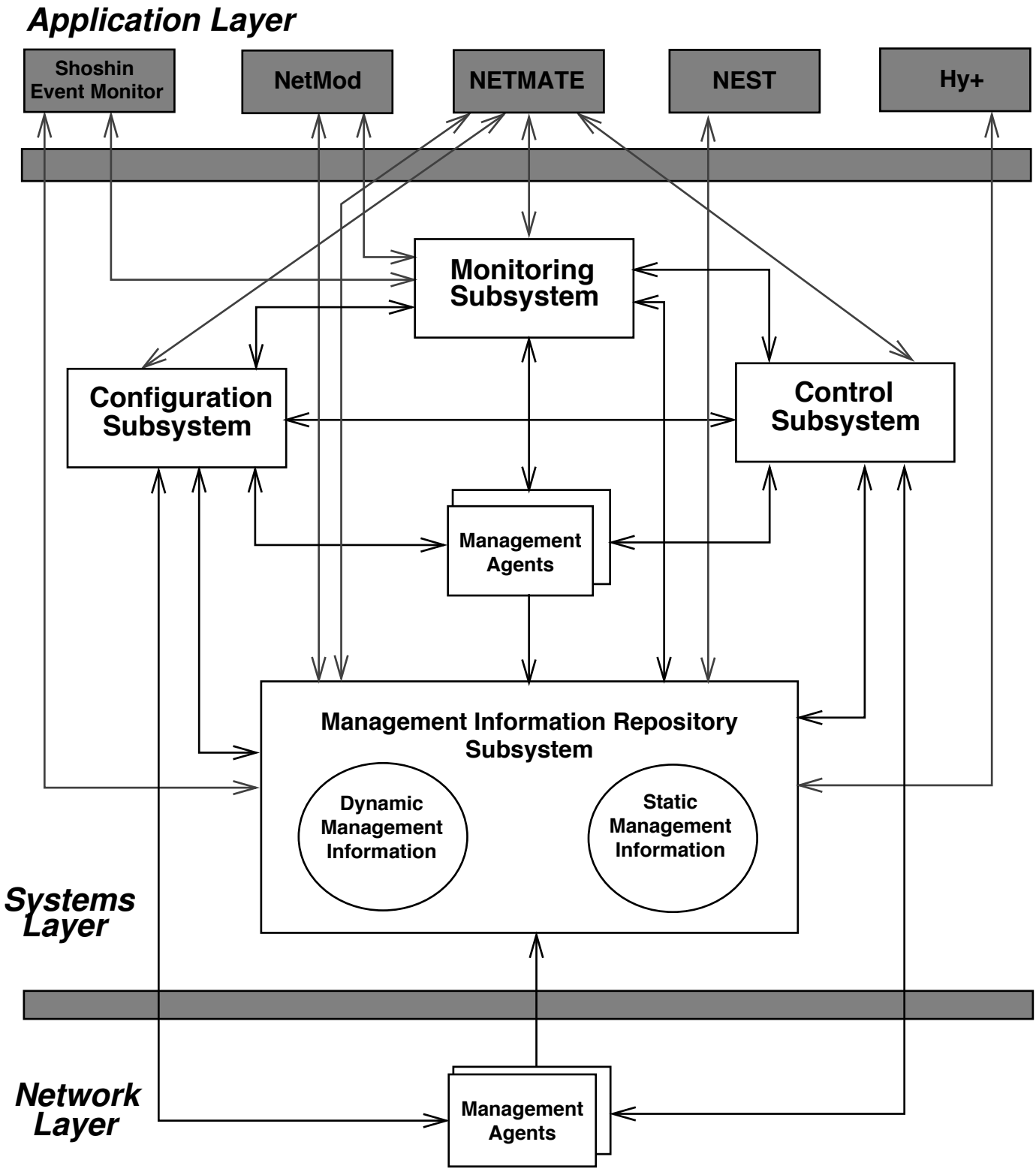

Figure 3. Manager/agent relationship

coordinating and synchronizing the activities of management agents, and executing monitoring requests.

- **Control Subsystem.** This subsystem is responsible for controlling the behavior of managed objects. For example, exceptions from managed objects may trigger control activities or control commands may be issued from the manager or administrator. The subsystem will ultimately control managed objects through interaction with management agents.
- **Management Agents.** The collection of management agents in the system and network layer that are dedicated to the control and monitoring of managed objects.

The network layer consists of the network hardware, resources, and services to be managed. The management agents are mostly the agents defined by SNMP and CMIP as an interface to managed objects.

3. Information Repositories: General Characteristics

This section discusses characteristics of the information repository.

3.1 Data

In order to address issues such as choosing a data model, we must first identify the types of data within the repository. Data from management agents (e.g., SNMP or CMIP) will comprise part of the repository information. One possible use of this data will be to track information about a particular object over a period of time, for trend or historical analysis.

We also expect configuration data, for example, data about the topology of the network, operating system version, and hardware specification. This information is used by application-level programs for operation, analysis, and presentation to the user.

A final source of information comes from application-level tools. For lack of a better term we call this data analyzed data. Referring to the CORDS architecture in Figure 3, the management tools will input information from the repository. The data could be recalled later and used as a starting point for additional processing or possibly for use by other management tools.

3.2 Requirements

We consider the following a starting point in listing the essential features of a repository.

- **Security**. A method needs to be in place that regulates access to information. Some information, for example, should only be accessed by system administrators.
- **Replication**. Replication is needed to insure availability, reliability, and performance.
- **Access interface**. The interface should be simple with rich functionality that includes add, delete, update, analyze, and display capabilities.
- **Naming scheme for objects**. Because the environment will be distributed, the naming scheme should be general, flexible, and allow consistent reference from anywhere in the environment, i.e., a global naming scheme. Also, the naming scheme should allow for autonomy of local sites, enabling them to delete and create new objects/names.
- **Adequate performance and scalability**. A repository must provide service at acceptable performance levels and in general must scale well. If the service is slow or cannot tolerate a larger community of users, as is common in many distributed environments, then it is not acceptable as a viable management tool.

3.3 Data Model Features

This section lists data modeling features that the repository should possess. We must keep in mind that the repository will be distributed and very likely heterogeneous. For example, some information can be stored in UNIX files (flat file format), while other information can be stored in databases (rela-

tional, O-O format). So the following features need not necessarily appear simultaneously in a single platform; rather we expect these modeling features to be present collectively in the entire repository.

- **Extensibility**. The model should accommodate unanticipated or future management needs. Objects will vary in type and size, and the objects themselves can change or new ones can appear. Therefore, an extendible data model is required to allow for flexibility.
- **Rich typing system**. It is desirable to allow attributes of objects to be user-defined or abstract, in addition to integer and string types.
- **Logical data independence**. The data model must allow the conceptual structure of the information store to be modified without changing the underlying storage structure. This allows different users to view the data in different ways. For example, a system administrator and project engineer have different needs and therefore view the data in different ways.
- **Composite objects**. Aggregation of objects into composite objects is a desirable feature. This would allow copying, deleting, and concurrency control of objects in an efficient manner—efficient in minimizing network communications and offering a simplistic view for the user.
- **Procedure as attributes**. This would allow virtual attributes (i.e., attributes that are computed by procedure rather than stored) that can strengthen the modeling capability.
- **Inheritance**. This simplifies the schema by reducing the number of data types and is a natural way to model many complex objects.

4. Experimental Prototype

Section 3 discussed the general characteristics of an information repository. We continue our study in this section by describing an experimental prototype. Our prototype consists of an application-level process, an information repository, and a network management agent. The network management agent will collect management information and store the information in a repository. The application-level process will access the data from the repository for processing and act as an interface to management information. We can perform experiments on the prototype to compare different repositories. Referring to Figure 3 in section 2, we will use XNetMod as our application level process, the Berkeley Packet Filter [11] as the management agent for collecting the data, and two candidate repositories for storing the management information. Our experimental prototype takes a vertical slice from Figure 3 and serves as the motivation for the remainder of our study.

The next section introduces XNetMod, giving a brief overview of its use, and a review of the data it will store to the repository. After introducing XNetMod, we conduct performance experiments to compare two candidate repositories, AFS and X.500.

4.1 NetMod Data Requirements

XNetMod [16] (Network Modeler) is a network performance tool developed at the University of Michigan's Center for Information Technology Integration. XNetMod is an X Windows System application that allows the user to compose arbitrary, interconnected, LAN networks in order to analyze performance. The performance parameters include packet delay and utilization; both are based on a set of analytic models. The analytic models use a traffic stream as input. The stream consists of a triplet of mean packet rate (packets/second), mean packet size (bits/packet), and second moment of packet rate distribution. This traffic stream, or triplet, along with the topology of the network, constitutes the data requirements for XNetMod.

When users define networks for performance analysis, they must specify the traffic stream. This can be done in one of two ways. First by supplying them arbitrarily, typing them in from the keyboard; second, by selecting

"canned" traffic streams representing different user profiles, provided statistically by XNetMod. We have added a third method of stream specification. This third method senses or sniffs the network directly to obtain the traffic packets per second. We will store the packets per second information into the candidate repositories and evaluate the use of this information in our experimental prototype.

4.2 Storing Network Data

To actually monitor the wire, we used the Berkeley Packet Filter (BPF) [11] in promiscuous mode. An XNetMod agent processes the output from BPF and stores the data in either a UNIX file (AFS) or X.500. Figure 4 illustrates the situation.

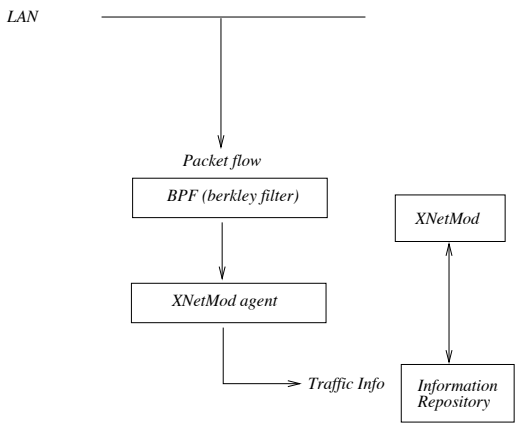

Figure 4. Manager/agent relationship

An interesting issue arises regarding the XNetMod agent: how often should the agent write to the repository? It is desired to supply the number of packets per second (pps) BPF is sensing on the network. Should the XNetMod agent count packets over a half hour interval and divide to compute pps? Or, should an entry be written to the repository each second?

To assist in this decision we conducted the following experiment. We set our XNetMod agent to write to the repository each second, over a 48-hour period. We call this sampling interval the flush interval because the agent writes or flushes the value of the packet counter to the repository once per interval. From this raw data, we derived the sample mean, sample variance, maximum and minimum samples, and several other statistics over varying flush intervals. For example, to derive the statistics for a 100-second flush interval, it is a simple matter of grouping or summing together 100 of the raw samples. Table 1 shows the results of this experiment.

Table 1. Derived statistics from packet counts over varying flush intervals

Flush Interval (sec)	σ/μ	Max. (pkts)	μ/sec (pps)	Max/sec (pps)
1	1.06	2074	104	2074
10	0.97	6960	104	696
100	0.92	51389	104	514
1000	0.87	442651	104	443
2000	0.83	790714	104	395
5000	0.77	1564031	104	313
10000	0.70	2775162	104	277

The Max/sec statistic is simply the maximum sample packet rate normalized over a 1-second flush interval. The variable μ is the sample mean. The μ/sec statistic is the sample mean normalized to a 1-second time interval. The variable σ is the sample standard deviation.

Let's analyze the σ/μ statistic first. As the flush interval increases, the σ/μ statistic decreases. This is to be expected because if we take any group of numbers and take larger and larger samples for each data point, then σ will tend to decrease, i.e. vary less and less [12]. We also know that network traffic tends to be bursty, so increasing the flush interval may tend to mask this burstiness. σ/μ behaves as we would expect and we do not see a significant dropoff until about 5000 seconds. The Max/sec statistic clearly shows that the burstiness drops off significantly at 10 seconds. So we conclude that merely counting packets over a long flush interval and then averaging to get pps will tend to mask the burstiness of the traffic to a significant extent. Coming back to our original issue of determining a flush interval for the

XNetMod agent, we choose 1 second. In our prototype version of XNetMod we will program the agent to store data for a 48-hour period and then begin overwriting samples. This will put a cap on the amount of space to be used. A future enhancement is to time stamp the data to a log file for trend analysis at a later time.

5. Experimental Repositories: X.500 and AFS

The following sections examine two candidate repositories: X.500 and AFS. We analyze each candidate in the following way. We list the salient features and compare these features to the minimum features a generic data repository must have. For example, a generic repository must have a method of naming and locating the objects within the repository. A platform needs to have the minimum requirements of the generic repository or it cannot be considered as a repository for XNetMod. Besides an analysis of minimum requirements, we also perform some modest measurements on both platforms. The goal of these measurements is to gain some insight on the issues of performance and scalability.

In a typical management system one can imagine sensors (e.g. SNMP agents), distributed throughout the system, gathering and storing data to the repository. We also expect various users and management programs to access the data. It is important to measure each platform to gauge the performance as multiple users and writers access the repository. We perform some initial experiments on each repository in an effort to explore this issue.

The client machines used in the experiments are all IBM RS/6000s running AIX 3.1 or 3.2. The LDAP server is an IBM RS/6000 running AIX 3.2. The DSA is a SUN Sparcstation and the AFS file server is an IBM mainframe.

5.1 The AFS Distributed File System

AFS is a UNIX-based distributed file system. Development of AFS began in 1983 at Carnegie Mellon University, funded by IBM, under the name of the Andrew project [13]. AFS has gained popularity in recent years and its successor, DFS, is the distributed file system for the Open Software Foundation's Distributed Computing Environment (OSF/DCE). AFS is based on the client/server model. A high-level view of the AFS distributed directory model is shown in Figure 5.

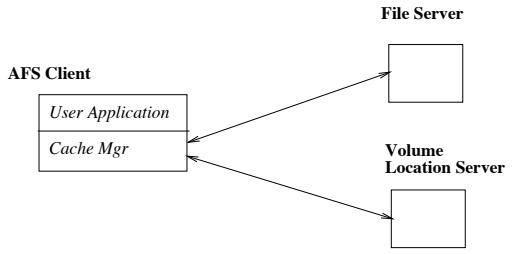

Figure 5. AFS distributed directory model

The main components of AFS are the cache manager, the volume location server, and file server. The cache manager runs on the client and accepts requests from the user application. As its name implies, it is responsible for caching on the client's local disk, and also for directing the processing of requests from the user application. The cell server and volume server maintain information about the location of cells and volumes, and the file server maintains its portion of the shared file system. Having a separate volume location server and cell server reduces the need for calls to the file server. When a user application performs a read, the cache manager checks first to see if the request can be satisfied on the local disk cache. If the data is not available locally, then the cache manager must satisfy the request from the file server.

A key design choice the designers of AFS made was to shift much of the workload to the client side and away from the server. Note that the cache manager on the client side is responsible for directing the entire process of satisfying client requests. The AFS file server does not take part in locating files;

the cache manager does that part of the processing. When the cache manager sends a request to an AFS file server, no other server need take part in the interaction because the server must have the data. This delegation of responsibility minimizes the amount of interaction with the file server and is a primary reason for AFS being highly scalable. Scalability is an important point to consider in choosing a platform for the information repository.

5.1.1 Features of AFS as a Data Repository

To evaluate the feasibility of AFS as a data repository for distributed systems management, we compare the main features of AFS against the requirements for an information repository that we listed in section 3.2.

Security is our first requirement. AFS uses the Kerberos [10] authentication scheme to validate users' rights to access files. The Kerberos authentication scheme has proven successful over the years. AFS also uses the standard UNIX file permissions in conjunction with its own set of AFS file permissions. AFS is also a *replicated* file system.

The *access interface* is simple with the basic functionality of add and delete. A possible disadvantage of AFS as an information repository is that it does not possess analyze and display capabilities. If AFS is to be used as a repository, it would be necessary to build a graphical user interface on top of it to provide these capabilities.

AFS is a global, hierarchical file system in which the name provides some information about file location. Thus, AFS provides an adequate *naming scheme* for objects.

So we see that AFS satisfies most of our list of basic requirements for an information repository for distributed systems management. In the next section we examine performance and scalability.

5.1.2 Assessment of AFS Performance and Scalability

In this section we collect some modest performance measurements on AFS. The goal of our first experiment is to determine how long it takes to perform a read, both from the cache and the file server. In this experiment we are trying to duplicate the situation in which a reader accesses or reads a single block of data and there is no other reader or writer contention. Table 2 shows our results.

Table 2. AFS read times

Server hit (sec)	Cache hit (sec)
0.0657	0.0005

We performed the read experiment over a 1-hour period, averaging one read per second. We performed the experiment twice, once with normal AFS caching and a second time with caching turned off. Under normal conditions, with caching on, the first read would come from the server and all subsequent reads would come from the cache. This behavior is satisfactory for our cache hit experiment. But for our server hit experiment, we wish to measure the performance of the AFS server. Therefore we needed to circumvent normal AFS caching and force all reads to come from the server. We achieved this by supplying an updated AFS library to the test machines. This updated AFS-allowed selective caching, i.e. either normal or no caching whatsoever [8]. The "server hit" value in Table 2 represents all reads coming from the server. Conversely, the "cache hit" value represents all reads from the local cache. As we would expect, the cache read time is much faster than a read from the server. A PING to the AFS server reveals a round trip time of approximately 6 ms, therefore the local processing time plus AFS server time is about 0.0507 seconds. We state the PING time to assure the reader that the network time does not dominate the stated server hit time.

Our next experiment makes a modest evaluation of how well AFS scales. With caching turned off once again, we attempt to duplicate the situation of multiple readers and one

writer. Consider for a moment the situation where a management agent is writing to the data repository and multiple readers are accessing this data item nearly simultaneously. Then, depending on how fast the writer is writing, we would expect most of the reads to come from the server.

By forcing all the reads to come from the server, it is as if we have an extremely fast writer. Each time the writer writes and closes, the cache manager invalidates the cache and the read comes from the server. We set up 3 different machines with our modified AFS so that caching could be turned off. The readers are modeled as Poisson processes with a specified mean arrival rate.

Figure 6 shows the results of the experiment. The x-axis in the graph gives the number of reads per second as seen by the file server, i.e., the total traffic generated by the readers. The y-axis gives the average time per single read. In this experiment we used three test machines.

Figure 6. Plot of # of reads per second versus time per read

Our experiment does yield some interesting results. We note that at 20 reads per second the read time is about 0.100 seconds. Which is about 10 times slower than a single read, but still acceptable from a performance standpoint. A point to consider is how often we would expect, under real circumstances, to be reading 20 times per second from the server. This represents a significant, but not implausible, workload. So we feel our initial results show that AFS seems to scale well under moderate work loads.

5.2 The X.500 Directory Service

We chose the LDAP (Lightweight Directory Access Protocol) as an interface to X.500. LDAP is on the Internet standards track. The LDAP interface to X.500 is easier to use than DAP and hides many details such as stack management and authentication. The LDAP/X.500 model is shown in Figure 7.

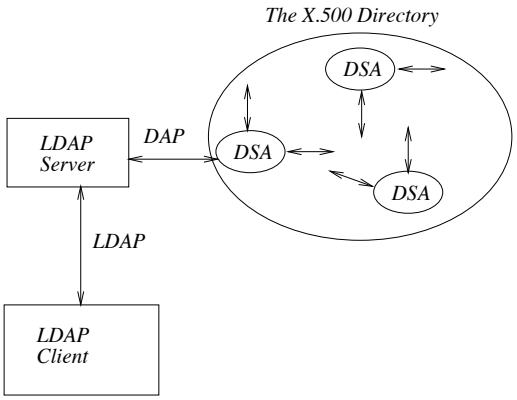

Figure 7. LDAP/X.500 directory model

To illustrate the features of the X.500 model, trace through the sequence of actions that occur when a generic user issues a "read" operation (Figure 8). The user issues a read request that is forwarded to the LDAP server. The LDAP server converts the LDAP read request into a DAP request and the server forwards the request to a Directory Service Agent (DSA). It is the responsibility of the DSA to actually locate the object within the directory. This may involve a chaining of DSAs until the object is actually located. A chain of DSAs occurs when the initially contacted DSA is unable to satisfy the request. In this case, the request is forwarded until the DSA that physically stores the X.500 object is found. Thus the user need not be concerned about the location of an object, nor how the object will be retrieved. The object is then returned in reverse order along the chain from the DSAs to the LDAP server. The LDAP server returns the object to the client to complete the operation. LDAP does not do any

caching unless it is explicitly told to do so. The X.500 standards do not specify caching algorithms, but the implementors of the DSAs normally do provide some caching.

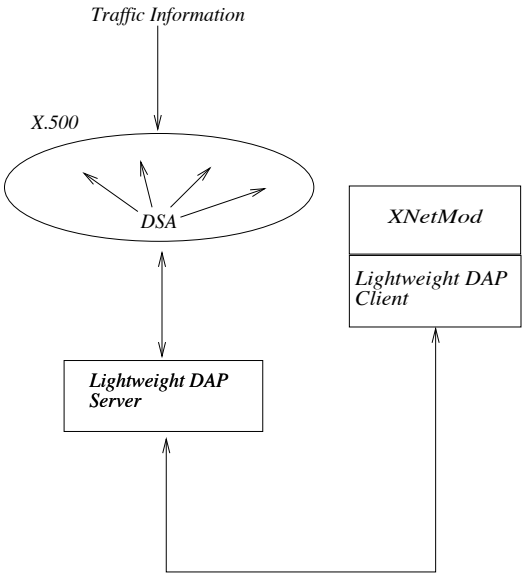

Figure 8. Information flow for an X.500 read operation

5.2.1 Features of X.500 as a Data Repository

To evaluate the feasibility of X.500 as a data repository for distributed systems management, we once again compare the main features of X.500 against the requirements for an information repository that we listed in section 3.2.

X.500 has acceptable security; it accepts Kerberos [10] and clear text passwords as methods of authentication.

X.500 has a friendly access interface that includes add, delete, modify, and lookup of objects. It does not possess an analyze capability so again, as in AFS, another layer of software would be required to do this.

A repository must have the ability to model arbitrary objects. X.500 was designed to store information about people, a "global white pages". However, X.500 also has a mechanism in which other types of objects can be defined. For the version of X.500 used in our experiments, Quipu version 7.0 [9], the hierarchical data model is used. Quipu version 7.0 allows object class definition, inheritance, mandatory attributes, and optional attributes. Therefore, X.500 allows various types of information to be defined, other than people objects.

A repository must have some scheme for naming and locating objects. X.500 provides a global, hierarchical, naming scheme to identify and locate objects. The entities are distributed and replicated, providing availability, and reliability.

Performance is also an important issue. The amount of time it takes to retrieve and store information is critical. Scalability is another factor. A repository will likely support many different management activities, and so it is important to gauge the behavior of X.500 as several different processes attempt to use X.500 simultaneously. We examine performance and scalability issues in the next section.

5.2.2 Assessment of X.500 Performance and Scalability

We performed the same experiments on X.500 as we did on AFS. Recall that we performed the experiment over a 1-hour period, performing one read per second, and taking the average. We performed the experiment twice, once with normal caching on, and a second time with caching turned off. Looking at the results in Table 3 we might expect the server read to be somewhat faster. The additional time is incurred by the LDAP server. The LDAP client must connect with an LDAP server, then the LDAP server connects with a DSA where the data is physically stored. Conversely, the AFS cache manager connects directly with the AFS file server. LDAP makes it easy for the client to fetch X.500 data, but also adds another layer of software between the client and the data.

The cache hit time is quite fast as we would expect.

Table 3. X.500 read times

Server hit (sec)	Cache hit (sec)
0.2089	0.0009

Because the server hit time represents DSA time and LDAP time, it is useful to give a time breakdown to analyze how time is being spent. Table 4 gives a breakdown of the server hit time.

Table 4. Server hit breakdown

non-DSA time		DSA time	
0.0817 sec	39%	0.1272 sec	61%

The stated time for the DSA time represents the time the read request was serviced by X.500. The non-DSA time represents the time spent by LDAP, network transmission time, and local processing time. Our results indicate that the time spent by the DSA is almost 2/3 of the total processing time.

Our next experiment involved multiple readers directing their requests to a single LDAP server with caching turned off. We increased the read rate of the readers and repeated the experiment several times. Once again, our goal was to get an idea of how LDAP and X.500 perform under increasingly large work loads. Figure 9 lists the results of the experiment. The key difference between this graph and the AFS counterpart is that an X.500 read time is higher than an AFS read at all read frequency levels. At 1 read per second, an X.500 read takes 0.24 seconds, while an AFS read takes 0.065 seconds. At a read frequency of 20 reads per second and beyond, X.500 takes 0.9 seconds while AFS takes 0.1 seconds. What is not clear at this point is whether the read times for X.500 are satisfactory for a real system. This issue must be pursued at a later time.

6. Conclusions

Because of the early nature of our experiments we wish to avoid harsh comparisons or make strong conclusions. Our mission at this point is to open some doors and make some general insights. Our data suggest that AFS scales better than X.500 and performs better through a range of work loads. X.500 offers better data modeling facilities that allow hierarchical data definition. So there appears to be a certain performance penalty in exchange for built-in modeling support.

Figure 9. Plot of # of reads per second versus time per read

7. Future Work

In this section, we discuss a number of related issues that outline directions for future work.

7.1 Repository Design and Data Model

The type of data model the repository is based on is an important issue. For example, how do we choose between relational, object-oriented, or hierarchical data models? Is it possible to build an adequate system which in based on one data model only? Or, are there advantages to be realized, such as performance, in adopting a heterogeneous approach? A heterogeneous approach is one that utilizes storage based on more than one data model. An example would be to store part of the information in flat file format, and storing more complex information in O-O

format. We mentioned that the repository would be distributed and that we required replication for performance and reliability purposes. Can we use existing distributed database or file system technology, or must something new be devised?

We must develop a methodology for the design and specification of information repositories. A key part of this design is how the data will be modeled. Included in this specification would be a list of applicable criteria that could be used in different situations as a means of selection. We must also define the role that modern database systems can play in the overall data management scheme.

7.2 Analysis of Existing Management Protocols

Earlier, we discussed the management protocols SNMP, CMIP, and SNMPv2. SNMP is strictly a network management protocol. However, CMIP and SNMPv2 allow objects to be managed in the system and application layer as well. The role of these protocols in the design of an information repository is unclear. In reference to the data requirements for the repository, can the objects defined by the protocols be used directly or must some other layer be built on top, possibly utilizing the protocols as service primitives? We are concerned with the modeling expressiveness, functionality, and utility of these existing protocols. It may be necessary to propose a new protocol if the existing ones are deficient.

7.3 Performance Measurement of Information Repository Platforms

We presented results concerning performance and scalability of X.500 and AFS as candidate platforms for an information repository. Our results would be more complete if we were able to give a breakdown of the service times to include local processing time, network transmission time, propagation delay, and server time. These results would be useful as a means to evaluate and compare the candidates. The open research question for this project is as follows.

Suppose we choose a platform and then proceeded to implement the system. If the end system does not perform as expected or does not scale well, then a poor choice has been made. Therefore some methodology should be developed to predict the performance and scalability of a candidate platform. A series of preliminary tests can be devised to weed out the worst cases. And then a series of more comprehensive benchmarks can be devised to expose the performance of the candidate system. A strategy to implement the tests would mainly involve measurements and tests on real systems. The reason for this is that it is not possible to develop a priori analytic or simulation models for any arbitrary platform. There are many good simulation tools that are quite flexible, but are sensitive to the amount of detail about the system that is programmed into the simulation and a knowledge of the service times and functions that are involved. On the other hand, a series of measurements and tests on a real system is flexible and can be applied to arbitrary systems.

8. Acknowledgments

We wish to thank James Hong for providing expertise in DAP and X.500. His advice was extremely helpful in building the LDAP client and server. We also wish to thank Tim Howes for revealing the internal structure of LDAP servers and for putting up with numerous amounts of email. Michael Stolarchuk shared his AFS expertise and was instrumental in installing the specialized AFS kernels on the test machines.

References

[1] ISO/IEC DIS 9596. "Common Management Information Protocol." October 1988.

[2] D.W. Bachmann, M.E. Segal, M.M. Srinivason, and T.J. Teorey. "NetMod: A Design Tool for Large-scale Heterogeneous Campus Networks." *IEEE JSAC*, 9(1):15-24, January 1991.

[3] M. Bauer, P. Finnigan, J. Hong, J. Pachl, and T. Teorey, "An Integrated distributed Systems Management Archtiecture." Submitted to the 1993 CAS Conference, October 1993.

[4] J.D. Case, J.R. Davin, M.S. Fedor, and M.L. Schoffstall. *The Simple Network Managment Protocol*. Internet Request for Comments 1067, August 1988.

[5] M. Consens, M. Hasan, and A. Mendelzon. "Debugging Distributed Programs by Visualizing and Querying Event Traces." *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993.

[6] A. Dupy, S. Sengupta, O. Wolfson, and Y. Yemini. "NETMATE: A Network Management Environment." *IEEE Network*, October 1991.

[7] D. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. "NEST: a Network Simulation and Prototyping Testbed." *Communications of the ACM*, 33(10):64-74, October 1990.

[8] T.J. Hacker "The Design and Implementation of an AFP/AFS Protocol Translator." CITI Technical Report 93-5, University of Michigan, August 1993.

[9] S.E. Kille. *Implementing X.400 and X.500: The PP and QUIPU Systems.* Artech House, Boston MA, 1991.

[10] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. "Authentication in Distributed Systems: Theory and Practice." *ACM Operating Systems Review*, 25(5):165-182, 1991.

[11] S. McCanne and V. Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." *USENIX Conference Proceedings*, pages 259-269, January 1993.

[12] P.L. Meyer. *Introductory Probability and Statistical Applications*. Addison-Wesley, 1970.

[13] M. Satyanarayanan. "Scalable, Secure, and Highly Available File System for a Distributed Workstation Environment." *IEEE Transaction on Computers*, 39(4):45-67, April 1990.

[14] J. Slonim, M. Bauer, P. Finnigan, P. Larson, A. Mendelson, R. McBride, T. Teorey, Y. Yemini, and S. Yemini. "Towards a New Distributed Programming Environment." *Proceedings of the 1991 CAS Conference*, pages 155-172, October 1991.

[15] D. Taylor. "A Prototype Debugger for Hermes." *Proceedings fo the 1992 CAS Conference*, pages 313-326, November 1992.

[16] K. Deboo. "XNetMod: A Design Tool for Large-scale Networks." CITI Technical Report 93-6, University of Michigan, August 1993.