ASH-3: A DISTRIBUTED FILE SYSTEM OVER DISTRIBUTED SHARED MEMORY

Hani Jamjoom, Steve Raasch, Andrew Shih

Department of Electrical Engineering and Computer Science
University of Michigan
{jamjoom, sraasch, ashih}@eecs.umich.edu

April 21st, 1998

ABSTRACT

The use of distributed file systems (DFS) has been popularized by systems such as AFS and NFS. They allow users to share system resources from any location in the network. They also increase the reliability and availability of these resources in that underlying changes are transparent to the user. Concurrently, much research has been undergoing in the area of distributed-shared memory (DSM). The concept of DSM is similar to that of DFS, in that it can allow many users to share a common base of resources. In DSM, however, the resource is not disk storage, but rather memory. Like DFS, DSM allows all users to have the same view of the resource that is transparent when moving between machines. Because these two systems are so similar, we combine the power of a DFS with the ease of development available from a DSM system. We quantify the performance lost by using the DSM layer and will also demonstrate the gains in ease of development.

1 INTRODUCTION

The benefits of a DFS, such as AFS, lie in the ability of a large distributed network of users to share common information resources. With a DFS, files can be located on a few central machines (or distributed broadly), but will appear to the user as being a local file. This transparency allows easy integration with the local file system and ease of use. In addition, location transparency is possible because data locations are not defined by the naming scheme. Data is allowed to migrate between servers without large difficulty. In this way, reliability and availability can be maintained by adding additional resources or by migrating clients when resources are unavailable.

Unfortunately, nothing comes freely. Having a DFS generally means that there could be multiple users sharing the same file. In this case, it is necessary to maintain some form of consistency between access to shared files. There is an overhead for maintaining consistency, which includes design complexity. In general, messages have to be sent back and forth to get the location of data, fetch the latest copy, and maybe invalidate or update the contents of memory.

The idea of DSM is not very different from a DFS. DSM allows a network of users to share a common memory space. This memory space, while distributed, appears to be contiguous and local to the user. This system also maintains location transparency as memory can belong to any of the connected systems. The DSM library functions will maintain all of the data synchronization, caching, and locking that is required to maintain a consistent memory space.

We can make use of the DSM abstraction to take care of most of this work associated with consistency in a DFS. This way, building a distributed file system is simplified. The distributed-shared memory will act as a global file cache while automatically taking care of data consistency. Memory updates, for instance, would be unnecessary because the DSM would handle synchronization for that memory location.

Another important aspect to this system is scalability. By using a DSM, the ability to scale may be hindered to some degree. We attempt to minimize any such degradation in performance by relaxing the consistency requirements to the minimum required to maintain file integrity. In addition, we try to minimize the amount of traffic necessary at the servers, both in terms of network traffic and disk accesses.

2 GOALS

In designing ASH-3, our most important goal is creating a simple and lightweight architecture for implementing a distributed file system over distributed-shared memory. One important aspect of this design is location transparency of files. That is, files can migrate between machines without affecting the user's view of the overall system structure. This is particularly important in load balancing, replication, and failure resilience. Even though none of these aspects have been implemented, a good design should allow adding such extensions fairly easily.

Another important aspect of our design is caching and client operation. Even in a DSM approach, certain machines will act as the servers, containing the majority of useful files and information. These servers will therefore have a much larger amount of communication traffic than other machines. To minimize this scalability limitation, we offset as much work as possible to the client machine. By utilizing DSM, we can cache the files in shared memory. In this way, frequently accessed files need not be reread from the server, thus reducing the load on the servers.

We also measure the performance of our system in relation to traditional DFS. Because we are utilizing DSM as a separate package that is not an integral part of the DFS, we anticipate some reduction in performance. Section **7.2** provides a comparison between ASH-3 and NFS and also gives some insight on the network traffic that is generated.

3 FEATURES

In this section we give an external view of the system. That is, we present the system from the perspective of the end user. Where issues are closely related to the architecture of the system, however, we present them separately to allow easier understanding of the system in contrast to other file systems.

3.1 File System API

For portability and compatibility, we support a file and directory interfaces similar to that of UNIX operating system. This way, users need not modify their applications to be able to run on top of our system. Instead, they only need to relink them with the ASH-3 library. Of course, there might be some small behavioral changes that are due to differences in the semantics of file sharing.

3.2 Directories

We use a single name space for files that looks the same on all participating machines. This directory structure always resides in global memory. The main motivation for this is design simplicity. Currently, we back up the directory structure on one "root" machine. However, since this structure can grow to be both large and very critical, it might be important to divide it into smaller parts, or replicate it in its entirety so that it can be stored on different machines. Finally, we provide name transparency of the actual location of the file to the user. The directory structure must, of course, allow the system to know where each file resides in stable storage.

3.3 Semantics of File Sharing

In our design, we assume that concurrent file sharing is rare. We thus follow a weak consistency model. We implemented shared and exclusive mode locks. That is, we allow multiple readers and/or a single writer. The writers' updates are not visible to processes concurrently reading the file unless specifically requested be the user. In that case, all other copies of the file are invalidated and the latest update is propagated. This update usually occurs only when a writer closes the file. This is similar to session semantics, where updates only become visible after a modified file is closed.

4 DISTRIBUTED SHARED MEMORY

This project consists of two major parts: First, a DSM which runs on a number of workstations and provides the abstraction of a global address space, and second, a DFS which uses the DSM abstraction to provide users' applications with a global file system. In this section we will give a brief overview of the DSM system because of its importance in our project.

In DSM, the system looks identical to a traditional multiprocessor with a global space that is visible to all of the machines. Applications' communications and synchronization are done through this shared memory with the actual message passing that is required to implement the shared memory being transparent to the user.

Currently, Rice University's TreadMarks[1] is used to provide an efficient shared memory abstraction for implementing ASH-3. We chose this package because of its completeness and availability over a wide variety of

platforms. There are many important issues surrounding any DSM. However, we will limit our discussion to two main aspects of the system: the memory model and the synchronization primitives. These will have a big influence on both the design and performance of the resulting application.

TreadMarks uses a *release consistency* memory model. This means that explicit synchronization must be performed in order for modifications to shared memory to be reflected locally. Two primitives are used for synchronization: locks and barriers. The reason that TreadMarks follow this model is that constant or immediate updates will exacerbate the effects of the communication overheads associated with transferring the data from one machine to another. This is particularly undesirable in a false sharing situation. Using lazy release consistency, in particular, will allow the system to delay communications and batch the transfers, providing greater network throughput.

This shared memory abstraction will provide a convenient environment to build a distributed file system. With the simplification of data sharing and updates, most of the difficulties of constructing a DFS are alleviated.

5 ARCHITECTURE

We can divide ASH-3 into three software components: *Application Library*, *Client Daemons*, and *Server Daemons. (figure 1)*. These components address the three primary design issues in the system: communications, the directory structure, and data handling issues.

The application library calls allow applications to utilize the ASH-3 system by communicating with the client daemons. They are the interface to the system and are fairly similar to traditional UNIX calls. These calls communicate with a client daemon that exists on each machine. The client daemons are the first-level of the DFS. They are responsible for directory activities, moving file data into the application's address space, and requesting files from servers. Server daemons are responsible for serving files from stable storage. They are needed because we are assuming that the shared memory is not persistent, thus, the files may not always be available in shared memory. In addition, storing files on a stable store is necessary to ensure system recovery and reliability.

In designing the system, our initial goal was to utilize the DSM for communications and synchronization. After discussing the main components of our system, we will discuss how the directory structure and our data handling combine with the DSM.

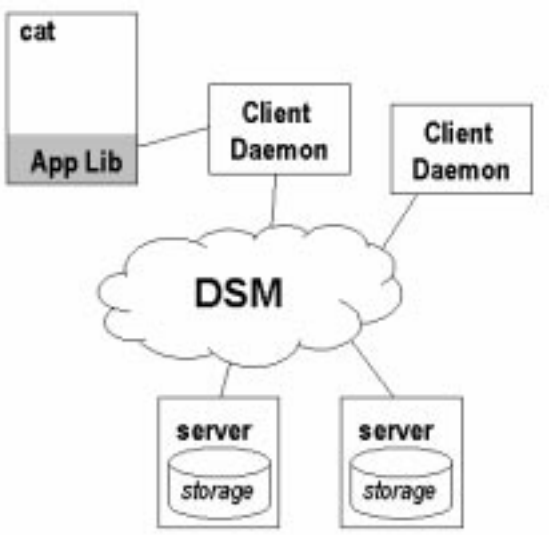

Figure 1: Architecture overview of ASH-3

5.1 Application Library

The role of the application library is to provide the necessary API for an application to communicate with the client daemon. As mentioned earlier, this API is similar to the one supported by the UNIX operating system. These functions allow the manipulation of both files and directories.

An application communicates with a client daemon via a local socket. Each call to the library moves the application's request into a packet and sends it to the client daemon for processing. It then does the reverse for the reply packet from the client daemon. An open call, for example, will send the name of the file, the open flags and the mode to the client daemon. The client will then reply with an open file descriptor to use for reading or writing to that file.

There are also some internal calls that the application library makes to the client daemon that are transparent to the user. These calls are responsible for making and destroying socket connections to the client daemon. A connection is made to the client on the first open request and closed on the last close request. This way, we do not need to establish a connection for every file request. We define a *session* as the period from the first open to the last close request. The application library keeps a count of the number of files that are opened by the application, when this count reaches zero, it destroys the connection to the client daemon which ends the ongoing session.

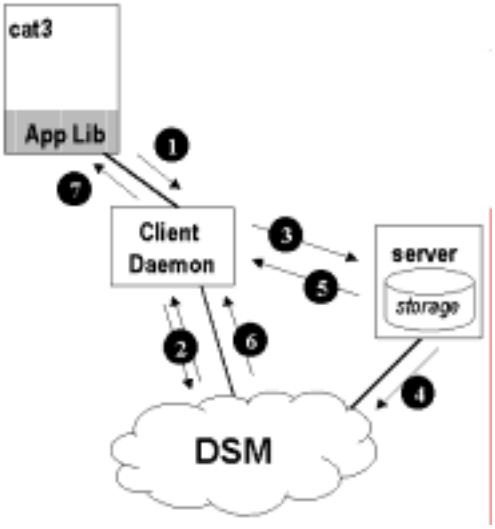

Figure 2: Example of a cache miss, i.e. the client daemon does not find the requested data in shared memory. (1) Application library makes call to client daemon (2) Client daemon does not find file in shared memory (3) Client sends request to appropriate server (4) Server puts data in shared memory (5) Server send client reply (6) Client reads data from shared memory (7) Client sends the application data

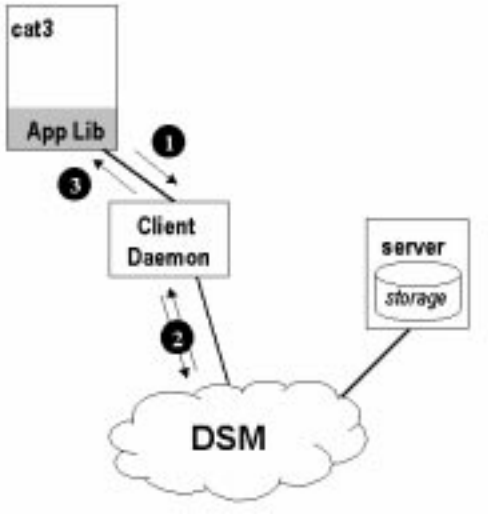

Figure 3: Example of a cache hit, i.e., client daemon finds the requested data in shared memory (1) Application library makes call to client daemon (2) Client daemon gets file if present in shared memory (3) Client sends data to application

5.2 Client Daemons

The client daemon pays two roles. It communicates with application to provide access to the DFS, it provides directory services, it moves file data from shared memory to the application's address space, and it also communicates with the server to retrieve files from stable storage. We will describe each part separately.

When an application initiates an ASH-3 request, it first establishes a connection to the client daemon. This marks the beginning of a new session. The client then initializes a *session table* for that application and returns a unique session ID that identifies the application to the client daemon. This table holds all of the necessary information about the connecting application, including the socket descriptor and a list of all of the files that are opened by the application. The information in the session table provides a layer of indirection from the file descriptors that an application holds to the location of data in the *opens files table*, which is a list of all of the open files in shared memory.

The second role that the client daemon plays lies in accessing the shared memory. Shared memory is used to hold data that the application needs. This includes all open files and the entire directory structure. Client daemons can access shared memory directly to read or write files. There are no explicit messages that have to be sent to achieve consistency in the system, because the DSM is designed to handle the synchronization.

5.3 Server Daemons

Server Daemons are simply client daemons with some added functionality. This way, we only need one server or one client daemon on each machine. An application that exists on a server machine can communicate directly to the server daemon instead of communicating with a client daemon. The added functionality that a server daemon has is the ability to receive file requests from other machines, and to move the requested files from its local storage into or out of shared memory.

5.4 Communication in ASH-3

We can think of the DSM as a global cache in which files are moved in and out according to the applications' needs. As mentioned earlier, files need not exist in the DSM all the time. Instead, they may reside on the servers' stable storage and are moved into shared memory when applications need to read or write to them.

Since we are viewing shared memory as a global cache, a *cache hit* occurs when the client finds the requested file in shared memory. In this case it can handle the application's request directly from shared memory. The problem comes when a *cache miss* occurs. That is, when a client daemon does not find the requested file in shared memory. In this case, the server that has the file in its stable storage will have to load the file in shared memory. Once that is done, the client daemon can access the file directly from shared memory to serve the application's request.

There are two options for dealing with the problem of asking servers to place files in memory. The first was to have each server poll on shared memory for such requests. This is a bad idea since it is not only CPU intensive, but also generates a large amount of unnecessary network traffic. The other alternative is to send a small message via sockets to the server requesting a file to be opened (or closed). We chose to implement the second solution. Though we are migrating slightly from using DSM for communication, it is clearly the better approach. Therefore, on a cache miss, the client daemon looks into the directory structure to find the server on which the file is stored (more about this later) and sends the server a small message requesting that it to place the specified file in memory. The server then reads the file from disk and places it in shared memory. The server replies to the client with a packet informing it that the file has been successfully placed into memory. The client can then serve the requesting application's needs. The cost of this forwarding of requests is two extra messages, one is the request from the client to open the file and another for the reply from the server. We only forward open and close file calls; once the server places a file in memory, clients can perform any read or write operation on shared memory.

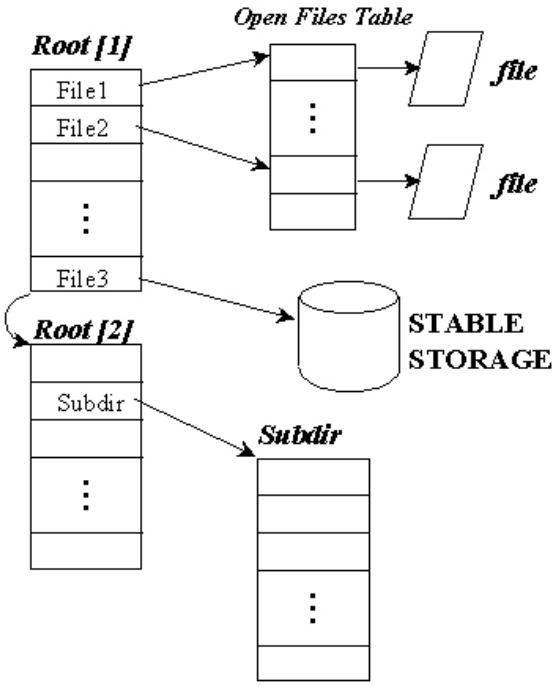

***Figure 4**:* Directory structure and Memory organization

One important thing to note is that clients do not block after sending a server an open file request. Instead, they continue serving other applications, thus improving throughput. The client continues to serve the requesting application when the server places the file in shared memory. Figure 2 illustrate the interactions that take place when a file is first read into memory. It shows the behavior of the system on a file cache miss. Once the file is placed in shared memory, figure 3 illustrates the behavior of the system for file cache hits.

5.5 Directory Structure

Probably one of the most important design aspects of the file system is the implementation of the directory structure. As mentioned earlier, one of our design decisions was to implement a file system that provides a single, global file name space. This simplified our design somewhat, because we could store a single structure in shared memory which would serve all users. Another decision is to allow any file to be stored on any server, and to allow files to be easily moved between servers.

Our solution is to maintain a unique global numeric file identifier and numeric server identifier for each file. A file's directory entry is accessed by traversing a tree consisting of nodes representing directory names, as is common for UNIX file systems. The directory node holds information concerning owner and group identifiers, file size and allocated space, and the server and file identifier. The file is retrieved from stable storage by contacting the appropriate server and requesting that it return the data associated with the file identifier. The server then finds the file on the host's native file system.

Using this approach, we can place all files for a server into a single host directory; the stable store does not need to be concerned with the directory structure of the ASH-3 file system. Since each file has a globally unique file identifier, any file can be placed on any server. The only bookkeeping necessary when moving a file is to update the server identifier field in the file's directory entry.

There are many different ways to allocate files to servers in a distributed file system, and our system can accommodate most any of them. For simplicity, we chose to associate a default server with each directory. The default server can be specified when the directory is created, or when the directory is moved to another server. When files are created in a directory, they are stored on the default server, unless another server is specified.

Our system does not currently have a way of maintaining a "current" directory, though one could be implemented by storing the current directory name in an environment variable, and pre-pending it to all non-absolute paths.

As mentioned previously, the file system directory is stored in shared memory. A global pointer is used to locate the root directory table. Each directory table is composed of eight table entries. Each entry has one of four states: *Empty*, *Sub-directory*, *File Entry*, or *Open File Entry (figure 4)*. An entry marked as a sub-directory points to a (possibly empty) directory table for the sub-directory. A file entry holds the server and file identifiers for the data on stable storage. An open file entry is identical to a normal file entry, except one field of the entry points into the global open files table. This table is used locate the file's data in shared memory and to keep track of the size and number of applications that are accessing it.

Since the file system directory is a shared resource, locks must be used to protect it from unsynchronized changes. In order to reduce the impact of one machine modifying the directory on another machine's ability to modify or read valid data from other sections of the directory, we decided to lock the directory at the granularity of a sub-directory. To properly support removing directory trees, a lock on a directory effectively locks all files and sub-directories below it. Unfortunately the TreadMarks DSM provides a limited number of locks. A file system using a fine-grained lock policy like ours requires a lock for each directory. This was clearly not possible with TreadMarks locks. Our implementation includes a "lock" field as part of each directory entry. Locking a directory sub-tree is accomplished in four stages: First, the entire directory tree is locked with a TreadMarks lock. Second, the directory is traversed from the root to the node we want to lock. If any node along the way is locked, then this lock attempt fails. Third, the entire sub-tree below the node is traversed to see if a lock is in place. If any node in the sub-tree is locked, the lock attempt fails. Finally, the directory node lock field is set, and the TreadMarks lock is released. The directory lock is released simply by clearing the "lock" field.

5.6 Data Handling

When file data is placed in shared memory, it is placed in a linked list of individually allocated blocks. Each block is made up of a fixed-size data area and a pointer to the next block in the chain. Since blocks are allocated one at a time, they can also be de-allocated one at a time. This allows us to grow or shrink the file's memory image easily. The linked list requires some effort to traverse, but this is not visible to the user, as the file access functions like `ash3_read()` and `ash3_write()` return a single contiguous block of data to the user.

6 PLATFORM

We have implemented this project over a cluster of Sun SPARC workstations since these machines are wildly available in the EECS department's labs. We also used TreadMarks as the choice for the DSM software package. This will run on the workstations to simulate the global shared memory.

7 PERFORMANCE

7.1 The influence of DSM on performance

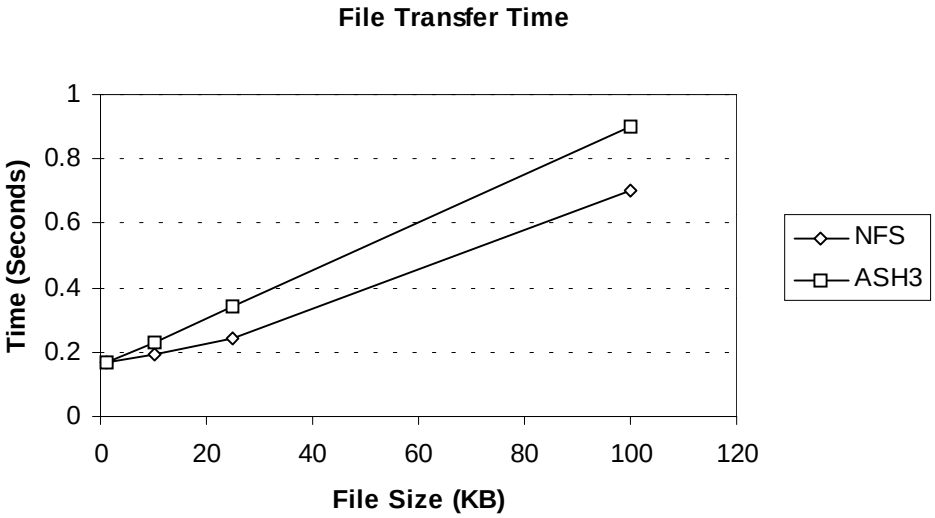

Figure 5: Average transfer latency for a remote read operation. Measured 1000 times and standardized for the average latency encountered.

Much of the design and performance of the ASH-3 file system is a result of the underlying TreadMarks DSM package. Briefly, TreadMarks is a user-level library that provides the DSM abstraction via message passing. It uses a lazy release consistency model to synchronize data between its client machines. The lazy release consistency model has been shown to allow TreadMarks to transfer fewer, larger packets than other consistency models.

TreadMarks uses an invalidate policy to force a client to demand-page modified pages back into memory. Invalidates are sent to a client when it acquires a lock that was held by another client. The client that most recently released the lock is responsible for informing the lock-acquiring client of pages that have been modified by it and other clients. The implicit assumption that makes TreadMarks' consistency work is that all modifications to shared data are protected by TreadMarks locks. Only if this is done will TreadMarks forward the proper invalidation information to the next client.

Our DFS system design is made more complex because it does not lend itself to the TreadMarks model. Examples are the fine-grained directory locking, access by several clients to disjoint portions of the same data structure (like the open files table), or multiple read-only clients (as seen when independent read operations are performed on file data). We solved the problem by placing `lock_acquire();lock_release();` pairs at critical points in the code. The effect of this was to cause all machines to be continuously updated with current information.

TreadMarks does, however help to minimize the amount of network traffic by computing deltas on pages of data in which multiple machines have made modifications. These deltas are shipped to the interested machines, where they are merged to form a single, coherent picture of the data.

Another point of interaction between TreadMarks and the ASH-3 DFS is in the page and buffer size. Since TreadMarks relies on the host machine's virtual memory system to trap access to shared memory, it also uses the same page size as the virtual memory system. The data block size used by ASH-3 has a dramatic effect on the amount of data that TreadMarks must pass in order to maintain consistency. The exact relationship is not yet understood, although tuning of ASH-3 has resulted in minimal overhead performance with a data block size of 1KBytes.

7.2 ASH-3 vs. Traditional DFS

To quantify the performance of our system versus a traditional file system, there are a few important aspects to measure. The first was to measure the time required to transfer files of varying sizes from a remote host. This test helps to indicate what level of delay the user will encounter when reading files from a remote host. In figure 5, we note that as file size increases, the added delay is approximately 20%.

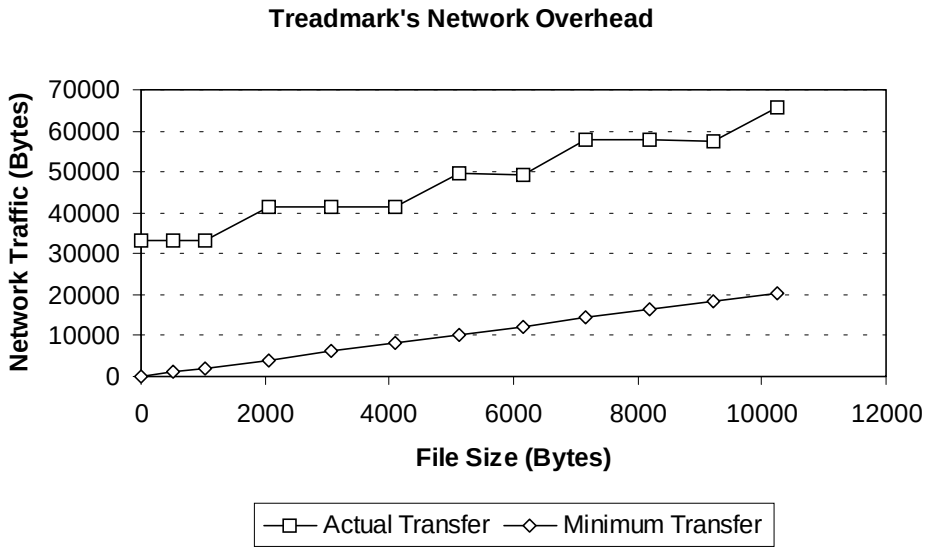

Figure 6: Network traffic overhead for a remote copy command versus the amount of data copied.

To move a file from a remote host to the local machine did involve some extra time, as more data handling is required by TreadMarks. In addition, the choice of file block size in ASH-3 is very important to the performance of the system. For instance, a file block size of 512 bytes will result in 33% less traffic than a file block size of 520. We believe this has to do with the internal handling of blocks by TreadMarks. With a block size of 520, TreadMarks was not able to place as many blocks within a TreadMarks page and more pages were transmitted.

To study the amount of overhead involved with using TreadMarks for data communications, we used a TreadMarks tool to display the number of bytes transmitted. This is a good indication of the amount of overhead involved in using the DSM. This overhead must be carefully controlled, as any increase can reduce scalability. To perform our testing, we used a completely empty directory structure and ran copy commands to a remote server. This requires reading a file from the remote server, reading it on the local machine, and then writing the file back to the remote location. For these tests, the minimum number of bytes transferred should be twice the actual file size.

From figure 6, we can see that there is about 30KB of overhead for copying any file. This is roughly the communications cost of distributing the open files and directory tables. Beyond that point, however, there does not seem to be much additional overhead for larger files. At 10KB, for instance, we need to transfer a minimum of 20KB of file data and 33KB for directory and open files information. That only leaves about 10KB of additional overhead. We believe that this overhead is the amount needed for synchronizing the memory pages within TreadMarks. On much larger files (512KB – 1MB), we noticed that that the overhead grew slightly (9%). For instance, with a 1MB file, there is a minimum of 2MB to transfer. Instead of the 10KB overhead from before, we noticed a 60KB overhead. This is because with larger files we are amortizing the fixed cost associated with maintaining internal control data over a larger number of data blocks.

8 Current Status and Future Work

At this point, we have fully implemented the application library and a set of simple utilities to verify and manage our system. Unfortunately, we have only been able to implement some basic tests on lightly loaded systems. The problem lies in the unavailability of workstations with enough available resources. Because of this, we only know that the system runs reasonably well with few clients. There has been very little stress testing of the system. It would be interesting to see how TreadMarks handles a heavily utilized shared space. In addition, we do not know if our current method of locking the directory structure and handling the open files table is easily scalable. With a heavy load on the system, there may be changes needed to relieve those two bottlenecks.

Another potential bottleneck that needs to be addressed is the amount of traffic at the servers. One way to do this would be to utilize some sort of persistent memory. This would allow us to reduce the strain on the stable store

and reduce that bottleneck. If it were then possible to hold everything in memory, some sort of load balancing could be implemented whereby all machines act as servers.

Future work also needs to be done in making this file system mountable. We must place wrappers around all of our system utilities so that files in ASH-3 are viewed as easily as are in NFS or AFS. With this in place, we will be able to run more complete tests to simulate real-life workloads. To give ASH-3 a fair test as a real DFS, we would also require that at least 128MBs of memory be placed on each workstation. With 64MBs of memory, there simply isn't enough free RAM to test the benefits of a global shared cache.

9 CONCLUSIONS

In the future, as memory prices continue falling and access to high-speed networks continues growing, the concept of distributed shared memory will only become more practical and achievable. With the DSM abstraction, much of the synchronization and caching needed to have a consistent and scalable distributed resource is already developed. To create a file system over such an abstraction reduces many of the difficulties in creating a DFS.

The main benefits are that the DSM handles file consistency, files can be cached in global memory, and scalability can be increased by removing server bottlenecks. There are, however, some limitations to using such a system, including additional overhead to the network, and required synchronization at the servers.

With the DSM model, we can use the calls provided to lock and synchronize our data. In this way, handling consistency was a matter of locating the best places to lock our system and to determine when syncs where necessary. For scalability, it is useful to treat the global memory as a global cache of file data and metadata. In this way, we can bypass the servers whenever possible, alleviating the bottleneck there. In fact, with a DSM that is tuned for file operations, it would even be possible to use it to do load balancing, replication of metadata, and as a smart cache. This would further reduce the workload at the servers, which tend to be the bottlenecks in a distributed system.

Along with these advantages comes the ability to utilize persistent memory as a form of stable store. While this would require even more memory for global storage, it would eliminate the need to have servers that require disk access. Because disks are much slower than memory and even the network, it would greatly help overall system performance if all files were available somewhere in shared memory.

With all of these benefits, however, are some fairly stiff limitations. First of all, there needs to be free memory at each workstation to serve as global memory. In order to have a good global memory hit rate, the amount of memory needed may be as high as 10% of the disk size. This is assuming that an average user will access around 10% of their files during normal usage. With file sizes growing all the time, the amount of memory needed to handle the data flow is obviously non-trivial.

In addition to the memory requirements, there is also an additional burden on the network. While the amount of traffic to the servers can be reduced, the overall network traffic will necessarily increase. This is due to the extra metadata that resides in shared memory as well as all of the synchronization that must be done. After any write operation, the openfiles table and directory table must be updated. This will involve updates whenever a subsequent user wishes to access a file. Fortunately, network bandwidth is increasing all the time. 10Mbit Ethernet lines are now moving up to the Gigabit range now.

With further testing and fine-tuning of our system, we will be able to get a better grasp of the benefits and limitations of TreadMarks. Even small changes to our file system structures can result in drastic impacts on bandwidth consumed and memory utilized. ASH-3 demonstrates that with the DSM abstraction, the complexity of creating a DFS is greatly reduced. The difficulty lies, however, in attaining traditional DFS speed with another layer of abstraction.

10 REFERENCES

[1] Amza, Cristiana, et. al. "TreadMarks: Shared Memory Computing on Networks of Workstations." Rice University

[2] Gozani, Shai, et al. "GAFFES: The Design of a Globally Distributed File System." University of California, Berkeley, 1987.

[3] Satyanarayanan, Mahadev. "Scalable, Secure, and Highly Available Distributed File Access." Computer May 1990: 9-21.

[4] Tanenbaum, Andrew S. Distributed Operating System. New Jersey: Prentice Hall, 1995.