Migratory File Services for Scientific Applications

John Bent, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny

Computer Sciences Department
University of Wisconsin, Madison

Abstract

We present the design and implementation of a migratory file service (MFS) for scientific applications on the Grid. An MFS is both easy to use and delivers high performance: two important properties that traditional Grid-based storage solutions are lacking. The key responsibility of an MFS is to dynamically create the user's expected file system environment on a remote cluster where the user's jobs are running.

We explore these concepts in a prototype MFS called the Hawk File Service. HFS transparently redirects application I/O to a set of migratory proxies that are located near the user's jobs; thus, the jobs can execute as if in their home environment. HFS also intelligently manages the set of proxies to account for the working-set size and failure characteristics of the cluster; thus, the jobs execute with faster access to their files and the overall throughput of the system is improved. Through a series of microbenchmarks and an application study, we demonstrate the effectiveness of the migratory file service paradigm.

1 Introduction

The Grid is a term used to describe a distributed computing infrastructure for advanced scientific and engineering researchers [9]. By exploiting Grid-based technologies, users can harness the power of large numbers of remote computational resources to solve what otherwise may be intractable problems [13]. Although originally targeted for a limited range of scientific applications, the Grid has evolved over the years, and recently has generated interest in the commercial sector [20].

Of central importance to a growing number of Grid applications is data storage. Commonly-used data sets are growing in size [1], and thus application throughput will increasingly be determined by the *performance* of the storage service they receive; if data throughput is insufficient, then application throughput (the true metric of success) suffers. Further, as more applications are developed within the Grid environment, *ease-of-use* of the storage system is of great importance; if users must expend a great deal of energy to manage their data sets manually, then the number of users is likely to be reduced.

Unfortunately, in today's Grid systems, storage is often treated as a second-class citizen. For example, many systems simply redirect all I/O operations from remotely-executing jobs to a "home" node, which is likely to be located across a wide-area link [19, 10]. Although easy to use, because the application encounters the same file system remotely that it does when running at "home", performance is sacrificed. To use local storage resources (and thus improve performance), users must instead manually load data onto the systems where they wish to run jobs [3]. In this scenario, the opposite problem arises: performance is gained, but ease-of-use is forfeited.

To obtain the best of both worlds and provide ease-of-use to users while delivering high-performance to applications, a new file service paradigm is required. Such a service dynamically recreates a users' storage environment in a remote cluster, enabling applications to run remotely with little or no human intervention. Such a service also provides excellent storage-system performance, exploiting local storage resources where possible. We term a system that creates such an environment for Grid applications a *migratory fi le service* (MFS), since it moves a user's file system environment to the site where the user's jobs are run.

To achieve this end, an MFS builds upon three key technologies. The first is common to all batch scheduling systems, namely, the core mechanism of *remote execution*; the MFS uses the batch system to initiate one or more *migratory proxies* on the distant cluster. The second technology is that of *interposition* [14, 16, 25]; once the proxies are active, the migratory service must then interpose on job request streams, redirecting client I/O requests to the proxies. The third technology is *multi-lingualism*; the proxies must be able to read data from and write data to the "home" file server, whether user data is stored on an AFS, NFS, HTTP, GridFTP, or other file server.

In building a migratory file service, we find that two important issues must be addressed. The first issue is of *performance management*. Specifically, the service must automatically tailor proxy performance to the workload's demands. In such a dynamic environment, there is no system manager to ensure things are working "well"; thus "no futz" system techniques must be developed [23]. The second issue of *failure management* arises because the migratory proxies are run as jobs in a batch-scheduling system. In such a system, nodes may be reclaimed at any time, as idle resources are reclaimed or as other jobs of higher priority enter the system. Thus, the set of migratory proxies must handle failure as a common-case scenario and not as a rare occurrence.

In this paper, we introduce a prototype migratory file service known as the *Hawk File Service* (HFS). HFS is built in the Condor batch-scheduling system [19], exploiting its job scheduling characteristics to run proxies and a remote file-service monitor on remote clusters. HFS utilizes Bypass [25], a low-overhead interposition agent, to redirect client I/O streams to the proxies. To converse with the home file service, HFS employs the NeST multi-protocol engine, thus embedding within HFS the ability to generate requests in a variety of common file-service protocols.

Built into HFS are two enabling features. First, HFS automatically adjusts the size of the proxy pool to reduce unnecessary file traffic between proxies and the home file server; the size of this active pool is dynamically based upon the working set of the user's applications. Second, HFS automatically includes excess servers in a "backup" pool to handle the frequent occurrence of node "failure"; the size of the backup pool is based on historical information for the average failure rate and recovery time of the cluster of interest. Crucial to sizing both the active and backup pools is the ability to monitor and subsequently adapt. Thus, HFS instantiates a file-service monitor on the remote site, which is responsible for both tracking the required performance and failure statistics and adjusting the size of the proxy pools.

We evaluate HFS through a series of microbenchmarks, demonstrating the behavior of the system under performance and failure scenarios. Overall, we find that the monitoring and adaptation mechanisms of HFS are effective. We also show that HFS can be used effectively by a real application through a case study of the BLAST DNA database searching program [2].

The rest of this paper is structured as follows. We give an overview of HFS in Section 2. In Sections 3 and 4 we describe two key pieces of functionality in HFS: the ability to size the active pool for performance the backup pool for availability, respectively. We then present application experience in Section 5, discuss related work in section 6, and conclude in Section 7.

2 Overview

In this section, we present an overview of our environment. We begin by describing a typical usage scenario for HFS. We then present our assumptions about the types of applications that will likely use HFS. Finally, we summarize each of the components in our system: user applications, remote data servers, a global scheduling system, proxies, and a local monitor that controls proxy behavior.

2.1 Example Scenario

The following is a typical usage scenario for migratory file services. A user has developed a particular application on their home system, and after debugging is complete, wishes to move to production mode. In this mode, the job will be run many thousands of times, likely over the same input data set, but while varying certain command line parameters. Through the Grid infrastructure, the user has access to one or more remote clusters of computers, and wishes to run the jobs at these sites (for simplicity, we will assume a single remote cluster is the target). Further, the potentially large input data set lies on the user's home file server. The difficulty occurs when the compute cluster and the data server are not in close proximity to each other. Although the user can submit the jobs using a Grid scheduling system and have each job access the data remotely (poten-

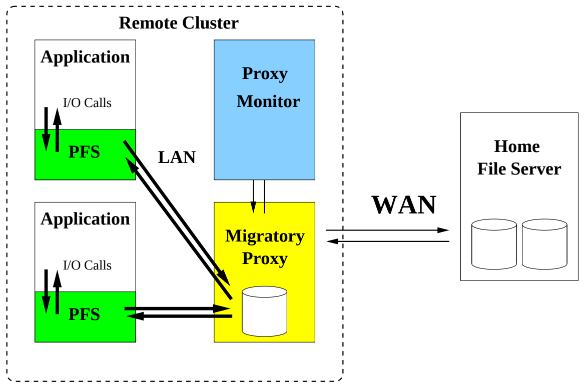


Figure 1: **Migratory File Service: An Overview** *The diagram shows the relationships between the home file server, the monitor, a single proxy, and two user jobs. Each application has its I/O calls redirected by an interposition agent to the local proxy, which then either serves the data from its local disk cache, or fetches the needed data from the home file server across the wide area. The monitor observes load and failure characteristics within the remote cluster, in order to determine how many proxies to instantiate to deliver high throughput.*

tially across the wide area), doing so greatly reduces throughput of the workload.

HFS is designed to eliminate this unnecessary latency by dynamically instantiating and managing a pool of proxies, which run on the remote cluster and interpose on I/O requests from the user's jobs. The proxies have the role of caching data near the user jobs such that latency is reduced for subsequent accesses to the data by this user. Proxies are submitted to the cluster in the same manner as the user's job; thus, to the Grid scheduling system, they appear as if they are normal user jobs. HFS also instantiates a Hawk monitor process, which controls the size of the proxy pool in order to maximize performance given current workload and failure characteristics.

This relationship across components is illustrated in Figure 1. A home file server is located in a remote location from the target compute cluster. Each of the four nodes in the available compute cluster is used for running either a user's job or a proxy; clearly, the more nodes that are used for running proxies, the fewer nodes are available for running user jobs. HFS also runs a monitoring process that observes the behavior of each proxy to determine if proxies should be added (or removed) from the system to improve throughput.

2.2 Typical Workloads

Many of the data-intensive workloads that are suitable for running on the Grid have similar characteristics. In particular, we assume the following, based on the findings reported by Foster and Avery [7].

- Many related jobs are often submitted by a given user at a particular time.
- Related jobs are likely to read the same input files, though the offsets within those files may differ. Total input file size may be from a few MB to on the order of many GB in size.
- There is little write sharing across files; that is, each job produces its own output file. Output files are often not as large as input files; output files on the order of a few KB are not atypical.
- Most jobs do not both read and write the same file. Rather, many workloads are set up in a pipeline, where the output of one phase forms the input of the next phase.

2.3 System Components

We now describe each of the components of HFS in more detail.

2.3.1 The Home File Server

The home file server can be any file server capable of running in a distributed setting (*e.g.,* NFS, AFS, FTP). Thus, the proxies in the system must be able to communicate with the home server in the required protocol.

2.3.2 User Jobs

It is important that the presence of proxies in the system is transparent to the end user (except for a improvement in performance). Subsequently, the user jobs submitted to the cluster must not require modification to leverage the proxies. Therefore, we rely upon an interposition agent to redirect I/O operations performed by the application to the appropriate proxy.

For an interposition agent, we use the Pluggable File System (PFS), which is built within the Bypass interposition system [25]. PFS adapts legacy applications to new storage systems by interposing on I/O-related system calls via hooks available within many dynamically-loaded library subsystems. In HFS, we utilize PFS to build a *application agent* that redirects relevant I/O calls to the proxies. The application agent also has other responsibilities, which we describe in later sections.

When the user's job first requests a file, the application agent opens a connection to the monitor and fetches the current list of proxies. The agent then uses the requested file path to determine the target proxy and opens a connection to it to service the request. These connections are persistent and subsequent mappings to the same proxy use the pre-existing connection. Whenever the contents of a proxy directory change, the monitor pushes the change to each connected agent. Before each read, the agent checks whether it has received any revised directories. Additionally, whenever an agent's request fails due to a lost proxy connection, the agent waits until it receives a revised directory.

2.3.3 Global Scheduling

A global scheduling system is required to find available resources in the compute cluster and to execute both the user jobs and the proxies within the cluster. A variety of batch scheduling systems exist to fulfill these purposes (*e.g.*, Globus [8], PBS [26], and LSF [29]). In our prototype, however, to schedule and execute jobs, we use the Condor [19] batch scheduling system.

2.3.4 Proxies

The migratory proxy is one of the major new pieces of software within HFS. Because no administration or human intervention can be expected, each proxy must be run on the compute cluster using the existing mechanisms in the global scheduling system. In Condor, this implies that the proxies are run as user-level jobs with no special privileges. The primary differences between a proxy and a typical file server or proxy are that each proxy only services a single user, does not have guaranteed storage space, and may be killed at any time. We briefly discuss the implications of these differences.

Private per user: Each proxy services only a single user; therefore, each proxy can be treated as an extension of the user's job. Specifically, the home file server trusts the proxy to the same extent as the user job. This implies that all authorization is the responsibility of the home server, and not the proxy. In our implementation, each proxy merely authenticates the identity of the user and forwards the request to the home server, which checks for appropriate access permissions; all authentication is performed with Grid Security Infrastructure (GSI) certificates [11]. This division of responsibility means that proxies cache not only files, but also the access permissions; a potential drawback of this approach is that changing the permissions at the remote server does not invalidate any copies currently stored by proxies.

No guaranteed storage: Because each proxy does not have guaranteed storage, the proxy serves only as a cache for file data. The other components of HFS make no assumptions about the size of this cache. In order to have a large local cache, we assume that the proxy is able to read and write from local disks on the compute node; however, in the worst case, proxies can use memory as their only cache.

In our current implementation, each proxy is responsible for caching data from files. Requests by user jobs for in-cache data are satisfied using the local copy; if the target proxy does not have the requested data, it asks the home file server for the necessary data. Space is managed in simple LRU fashion.

Finally, HFS does not guarantee file consistency across different proxies. Given that the applications in our target workload have no write sharing, we believe this is a reasonable simplification.

Non-dedicated resources: Because proxies and user jobs may run on non-dedicated resources, both proxies and user jobs may be terminated at any time. As stated before, an important goal is that this termination should be transparent to the user process. Condor does provide some support for this case: jobs that are terminated can be automatically resubmitted. This simplifies the responsibility of the monitor since it does not need to resubmit additional proxies if one fails. However, two complications still exist.

The first complication is that the user jobs interacting with this proxy must be notified of this failure. In this case, the user job must be redirected to a new proxy which will be responsible for caching these files. Given the functionality of the application agent, the complexity of this notification is easily hidden from the actual user application.

The second complication is that a proxy may be terminated when it contains the only copy of data written by a user application. In this case, HFS must maintain an important property: *After an application exits, its output fi les are guaranteed to reside on the home storage node.* This property is consistent with the rest of Condor in that user jobs may be killed and restarted without side-effects before completion, but once the job has successfully terminated, all output must exist (*e.g.*, this is assumed by higher-level schedulers on Condor such as DAGman [4]).

One option for dealing with potentially lost data is to kill the user application as well. This requires that the user job waits on `exit` for all of its file data to be flushed from the proxies to the home node. Although this approach is simple to implement within the our current framework and maintains the desired property, it has the severe drawback that a significant amount of work may be wasted if the job was running for a long time before the proxy terminated; additionally, the chances of success decrease as the number of proxies (and therefore the number of dependencies) increases.

An alternative option exists if a proxy is notified that it will soon be terminated and given time to evacuate its data. In this case, the terminated proxy can write its data either to the home node or to a backup node on the local cluster. If termination occurs before the proxy has written all of its data, this case reverts to the previous one. Although Condor includes functionality to notify upon termination, individual clusters may not be configured with this option. Therefore, we do not yet investigate this approach in our system.

A final option is for the proxy to always write through data back to the home node; the proxy waits until it has been notified that the write has succeeded on the home server before returning control to the user application. This approach has the disadvantage of potentially incurring more traffic on the home node and causing delays to the user applications; however, given that read traffic dominates write traffic in many of the scientific applications we are targeting, delaying writes may be acceptable [7]. Our current implementation takes this straight-forward approach.

2.3.5 The Proxy Monitor

The proxy monitor is the second major piece of software created for HFS. Because the proxy monitor interacts frequently with the proxies, the monitor should be located near the proxy cluster for the best performance. However, our current implementation assumes that the proxy monitor is reliable and, therefore, it should not be run on non-dedicated resources; this assumption implies that the proxy monitor may not always be able to be placed near the cluster containing the proxies or user jobs. In the future, though, we believe this restriction could be relaxed; because the monitor only consists of soft-state [12], it should be possible to tolerate its failure.

The proxy monitor has two primary responsibilities. First, the monitor must inform each agent of the address of each proxy and a small index used to map file blocks to proxies. Second, the monitor must determine the number of proxies that should be allocated to each user in each cluster. For simplicity, we divide the number of proxies into two components. First, some of the proxies are part of the *active pool*: this number is calculated such that the active proxies contain the working file set of the user. Second, some proxies are part of the *backup pool*: this number is set such that the size of the active pool is maintained on average, despite the unexpected termination of proxies. The mechanisms and policies required to support the active pool are described in Section 3, whereas those for the backup pool are in Section 4.

3 Performance Management: Sizing the Active Pool

The data set required by a single user job or a set of user jobs may be much larger than that which fits in the cache of a single proxy. In this case, the proxy will thrash, since it must evict local copies that will be accessed again in the near future. It is the responsibility of the Hawk monitor to manage the “active” set of proxies such that the current working set fits across the proxies. In this section, we describe the mechanisms and policies for this functionality and demonstrate the performance benefits of an active pool within HFS.

3.1 Mechanisms

Three basic mechanisms are required by the Hawk monitor in order to manage the active set of proxies. First, the Hawk monitor informs each application agent of the set of proxies in the active set and an index which can be used to determine which proxy is responsible for a particular file block. Second, the Hawk monitor determines the current performance level of each proxy; that is, whether or not the jobs working set fits in the current proxy caches. Third, the Hawk monitor adds (and removes) new proxies to the pool as needed for performance. We briefly describe these three mechanisms.

3.1.1 Informing Jobs Of The Active Set

The Hawk monitor is responsible for managing the list of proxies in the active set and notifying the client-side agent of the location of the proxies. When a user submits a new set of jobs to the cluster, a Hawk monitor is created as well and the user jobs are initially aware of only the location of the Hawk monitor.

The Hawk monitor has the task of starting each of the proxies in the active pool. The Hawk monitor maintains a persistent connection with each of the proxies, so that it knows immediately when a proxy fails. The Hawk monitor informs each of the client-side agents whenever a proxy enters or leaves the system. In the current implementation of HFS, the Hawk monitor is also responsible for determining which proxy is responsible for which user data. This mapping is performed in a centralized place since all user jobs should contact the same proxy for the same data to avoid duplicating data across proxies and wasting cache space. We describe our mapping function and how it changes as proxies enter and leave the system in more detail as a policy decision.

3.1.2 Obtaining Proxy Performance

The Hawk monitor is responsible for determining the correct number of proxies; this requires obtaining information from each proxy regarding their performance. In HFS, each proxy is responsible for recording this information about its performance and the Hawk monitor periodically queries each proxy to obtain this information.

3.1.3 Adding a New Proxy

After the Hawk monitor has determined that a new proxy should be added to the working set, the Hawk monitor asks Condor to start a new proxy on one of the nodes in the cluster. There is one piece of functionality that would be useful to add to Condor to support this requirement: preempting a lower priority job (*i.e.*, the user's job) in preference for a higher priority job (*i.e.*, the proxy).

When there are more proxies and user jobs submitted to the system than there are available resources, HFS would like proxies to have a higher priority than user jobs. Condor does contain mechanisms to allocate idle resources to high priority jobs, but it does not allow high priority jobs to preempt low priority jobs.[1] Thus, in the initial phase when proxies and user jobs are submitted at the same time, the proxies start execution first, as desired. However, if the monitor later determines that more proxies are needed to improve performance, Condor does not have a simple way to preempt the lower-priority user jobs for these proxies. Thus, Condor would wait until one of the user jobs finishes before starting the proxy. To remedy this, our monitor watches for this situation and explicitly asks Condor to suspend enough user jobs to allow the proxies to run; the monitor first vacates jobs with the shortest current running time so as to minimize the amount of lost work caused by these removals.

3.2 Policies

We now describe how HFS leverages these basic mechanisms for managing the active pool; specifically, we describe the HFS policy for mapping data across proxy caches and determining when a proxy should be added to the active pool (or removed) for improved performance.

3.2.1 Mapping Files to Proxies

The major challenge in designing a function for mapping files across proxy caches is adapting the map when proxies enter or exit the system. We have three goals in designing an algorithm. First, the mapping should balance the load of file requests across proxies. Second, the addition or removal of proxies should not radically alter the mapping since remapping data incurs a cost; that is, the remapped data must either be explicitly moved between proxies or the newly responsible proxy must incur a cache miss on the first request. Third, a newly added proxy should take over the load for those proxies that were the most overloaded.

[1]To be exact, in its current implementation, Condor will sometimes perform priority-based preemption, but only to run a higher-priority job of a *different* user.

To meet these goals, HFS uses an algorithm derived from extendible hashing [6], which is designed for graceful growth and reduction of the table size. The number of entries in the hash table is a power of two and is greater than or equal to the number of proxies in the active pool.

The hash table entry for a given file is calculated using the B least significant bits of its hash number. If a proxy corresponds to that hash entry, then the request is sent directly to that proxy. Conversely, if the hash entry is empty, then the $B - 1$ least significant bits are examined; this process continues until the responsible proxy is located.

Thus, when the Hawk monitor detects that an existing proxy is thrashing, a new proxy is added that splits the workload of the previous proxy. Assuming the previous proxy was in hash position X, the previous proxy will continue to handle position $0X$ while the new proxy will handle $1X$. Note that the number of hash table entries is doubled if the new proxy is added beyond the current size of the table.

This mapping structure automatically adjusts to proxies being removed, whether because the resources were revoked by the scheduling system or because the working set decreased. In either case, the load is automatically redirected to the parent node, just as if the removed node never existed. Note that this mapping assumes that the root of the hash table (*i.e.*, $B = 0$) contains a valid proxy.

Extendible hashing meets our goals as follows. First, the hashing function naturally balances load across the proxies. Second, the responsibility of no other files are migrated to a different server. Files which hash to an empty slot are remapped to the same proxy that served it in the previous mapping. Notice further that should a proxy fail, its requests are remapped to the parent proxy who may still have copies of those files. Finally, using this scheme, the monitor can add new proxies exactly where they are needed. This also allows for fast growth of the table as in the case when multiple proxies are detected to be thrashing, the monitor can submit multiple proxies and map each new proxy to help each thrashing one.

3.2.2 Sizing the Active Pool

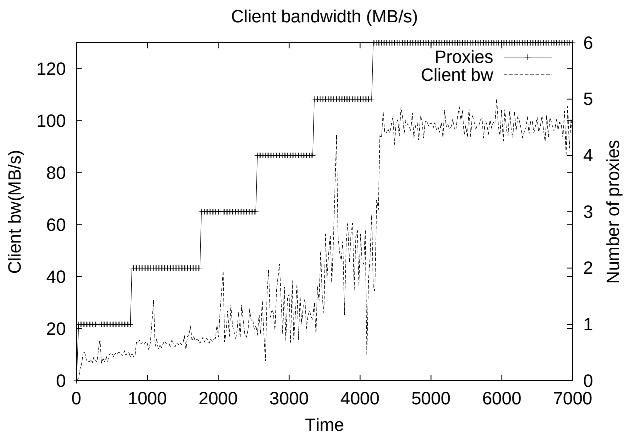
To determine whether a new proxy should be added to the active pool, the Hawk monitor must know whether each current proxy is able to handle the load directed to it.

In our current implementation, each proxy uses a modified approach to determine the relationship between the working set and the cache size; each proxy signals whether it is thrashing by reporting the number of *capacity* cache misses it observes (as opposed to cold-start misses) within an interval. When the number of capacity misses exceeds a threshold for some number of intervals, the Hawk monitor determines that a new proxy should be added to the active pool. The drawback of this implementation is that HFS cannot easily determine when a proxy can be removed from the active pool. However, in our environment with transient failures of proxies, this is not as large of an issue; when a proxy fails, HFS does not resume the proxy if it is no longer needed.

In the near future, we plan to implement signals from the proxy that correspond directly to the relative size of its working set. Specifically, the Hawk monitor will know the relationship between the size of each proxy cache and the working set that it is trying to handle. If the Hawk monitor observes that the working set directed toward a proxy is much greater than its cache size, then the Hawk monitor knows that either a new proxy should be added or part of its working set should be directed to another proxy. Conversely, if the Hawk monitor observes that the working set of a proxy is much less than its cache size, then the Hawk monitor may be able to remove a proxy and redirect its working set to this one.

3.3 Experiments

To demonstrate the effectiveness of these basic mechanisms and policies, we present the results of the following experiment. The synthetic workload that we consider here and in the following sections consists of 400 jobs, each of which accesses a randomly chosen 1 MB file 500 times (each job has access to 40 1-MB files, and thus there is likely to be some re-use). The user jobs and the proxy nodes are run on a Condor pool with 40 machines, and each node makes 10 MB of local disk cache available if a proxy is running upon it; to ensure we are running in a controlled environment, we do not allow other users to submit jobs to this pool. In this test, we also assume that the Hawk monitor is run near the proxy nodes. A home file server that contains



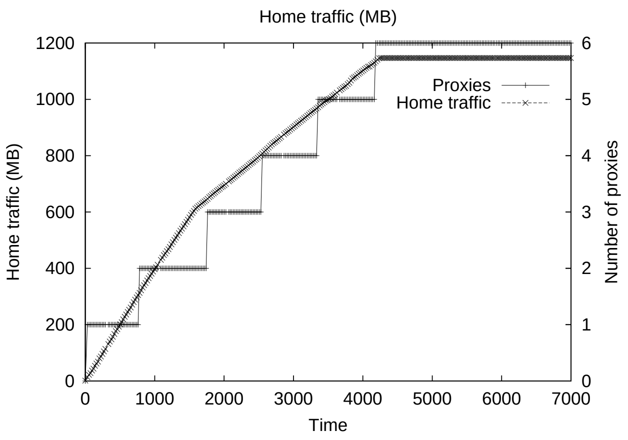


Figure 2: **Sizing For Performance.** *This experiment reveals the importance of correctly sizing the proxy pool in relation to the size of the clients' working set. The graph on the left measures the client bandwidth over time under the synthetic workload. As more and more of the working set fi ts in proxy caches, performance improves until all proxies have stopped thrashing. The graph on the rights verifi es that the algorithm has correctly identifi ed the size of the working set as the cumulative traffi c sent by the home server slows and then stops.*

the inputs files lives on a machine in a remote cluster. Within the cluster, a switch-based 100 Mbit/s network connects the machines; in addition, we control the performance of the link between the proxies and the home server, setting the emulated wide-area bandwidth to 0.5 MB/s and the wide-area delay to 100 ms. Note that this set-up is not meant to emulate any specific application; rather, it is primarily intended to bring forth the performance characteristics of our system.

The graphs in Figure 2 show that the bandwidth delivered to the clients increases as the Hawk monitor adds more proxies to the active set. Specifically, the graph on the left presents total client bandwidth as the number of proxies are increased. As one can see from the graph, as the system reaches 6 total proxies, the working set of the synthetic workload fits into the sum of the proxy caches. This conclusion can also be confirmed from the graph on the right, which plots the total traffic to the home file server; after 6 proxies are in the system, no more traffic to the home node is generated.

4 Failure Management: Sizing the Backup Pool

The number of nodes available to each user in the cluster can change quite rapidly, due to the arrival of jobs from a competing user, the reclamation of idle machines by their owners, or actual machine failures. Such "resource failures" may be quite common in the Grid, occurring much more frequently than the traditional rate of machine failures. Thus, determining the number of proxies in the system based on performance metrics alone is not sufficient – resource failures must be anticipated in order to maintain job throughput, particularly when the cost of starting a new proxy is high. In this section, we describe how HFS embellishes the "active" set of proxies with additional proxies in a "backup" pool such that high throughput is delivered to the user's jobs under the expected number of resource failures.

4.1 Mechanisms

We now discuss the mechanisms required to manage the backup pool of proxies. First, the Hawk monitor must have the ability to monitor how frequently nodes are failing (*i.e.*, determine the mean-time-to-resource-failure, or MTTRF). Second, the Hawk monitor must determine how long it takes to start a proxy (*i.e.*, determine the mean-time-to-resource-repair, or MTTRR). Finally, the Hawk monitor must have the ability to spawn proxies into the backup pool so as to anticipate future failures; since this mechanism is identical to the spawning mechanism presented in the last section, we do not repeat that discussion here.

4.1.1 Calculating MTTRF

To calculate the MTTRF of a cluster, the Hawk monitor observes failure rates of proxies and calculates a historical average. Observing a failure is simple due to the persistent connection the Hawk monitor maintains with each proxy; when the proxy undergoes a resource failure, the Hawk monitor notices almost immediately. By tracking when each proxy was launched and when it failed, the MTTRF can be readily calculated.

We note that the observed MTTRF may be sensitive to several variables; we consider two examples. First, the MTTRF is likely to be a function of the current time of day (*e.g.*, machines are less likely to be revoked during the night) and even the day of the week (*e.g.*, machines are less likely to be revoked on weekends and holidays). Thus, if the MTTRF is recorded and averaged over a long period of time, the phased behavior of the system must be taken into account. Second, the MTTRF is likely to differ across clusters (*e.g.*, some clusters may consist of desktop machines which will be revoked frequently by owners whereas others clusters may consist of resources dedicated to Grid computing). Thus, the MTTRF should be tracked separately for each cluster to which the user submits jobs.

Our current implementation does not perform sophisticated historical tracking. Instead, the MTTRF is obtained in one of two ways. First, the Hawk monitor can be configured at startup with an expected MTTRF for the target clusters; this approach is reasonable given that Condor collects similar statistics, but has the disadvantage that some manual intervention is required. Second, the Hawk monitor can track the MTTRF over its current lifetime; this approach is reasonably accurate given that users are likely to submit many jobs at once and thus each instance of the Hawk monitor observes the behavior of the cluster over a significant period of time.

4.1.2 Calculating MTTRR

To make an intelligent decision for the size of the backup pool, the Hawk monitor must also know the MTTRR. Defining MTTRR is not completely straight-forward since the time to recovery can be defined in one of two ways: the time from which the Hawk monitor submits a new proxy to either the point when the proxy begins execution or to the point when the cache state of the new proxy matches that of the failed proxy. Since the cache state of the new proxy may never match that of the failed proxy (*e.g.*, the working set changes or different traffic is directed to this proxy), for simplicity we consider only the former definition.

Using this definition, the MTTRR is easy for the Hawk monitor to monitor. Whenever the Hawk monitor submits a new proxy, it records the time of submission; when the proxy later begins execution it will contact the Hawk monitor to establish a persistent connection. The Hawk monitor calculates the MTTRR as the difference between these two times.

The Hawk monitor must average multiple observations of MTTRR in order to obtain a reasonable estimate, due to the fact that the time to launch a job may vary widely. For example, in Condor, a process known as "match making" [21] is employed to decide where to run which jobs. Jobs and machines are matched to one another based on the desires of the jobs (*e.g.*, a job needs a machine running Linux 2.2 or greater with 200 MB of memory) and the abilities of the machines (*e.g.*, a machine is running Linux 2.4 and has 1 GB of memory). This process of "negotiation" takes place at periodic intervals (perhaps every 10 minutes) and requires more time as more jobs are submitted to the system. Thus, if a job is submitted just after a negotiation has occurred, the job must wait the full cycle time to enter the system.

Although our experience lies primarily with the Condor system, the method in which the Hawk monitor obtains MTTRR is generic and should work across a range of systems. As mentioned above for MTTRF, maintaining historical information in a more sophisticated manner may be worth pursuing in future work.

4.2 Policies

By obtaining the MTTRF and MTTRR, the Hawk monitor can calculate how many proxies to launch so as to deal gracefully with anticipated failures. We now describe the policies used in handling resource failures within the cluster. Specifically, we describe the process of sizing the backup pool based on current resource-failure information, and how the backup pool is used when there are no failures.

4.2.1 Sizing The Backup Pool

To size the backup pool, the monitor considers the MTTRF and MTTRR to ensure that sufficient proxies exist when failures occur at the expected rate and recovery takes the expected amount of time. Intuitively, to keep the number of proxies in the pool constant, the rate at which proxies exit the system (*i.e.,* fail) must be equal to the rate at which proxies enter the system (*i.e.,* recover). Thus, the number of proxies in the backup pool must be equal to the ratio of the MTTRR to the MTTRF of any proxy. Given $MTTRF_{Single\ proxy}$, then the expected $MTTRF$ of some proxy in the system is $\frac{MTTRF_{Single\ proxy}}{Number\ of\ proxies}$. In summary, the backup size is calculated by the monitor as follows.

$$Size_{Pool} = \frac{MTTRR \cdot (Number\ of\ proxies)}{MTTRF_{Single\ proxy}} \tag{1}$$

This calculation does make a number of assumptions; namely that failures, as well as recoveries, are independent and identically distributed across proxies. Unfortunately, these assumptions are not likely to be true in our environment. First, it is likely that failures will be correlated; for example, at the beginning of the work day, it is likely that multiple machines will be revoked by returning owners. Second, and more dramatically, the time at which proxies recover is also likely to be correlated; given the periodic interval at which Condor runs the match making service, multiple proxies may be started nearly simultaneously. Despite these caveats, we feel that calculating the size of the backup pool using this approximation is an acceptable approach for a first step.

4.2.2 Backup Pool Behavior

The other policy question we must answer is how the proxies in the backup pool are utilized before a failure occurs. One option is for the backup proxies to not participate in any HFS service, and to instead run user jobs until they are needed to serve data. However, this option is difficult to implement, since many batch-scheduling systems do not allow more than a single job to run on a uniprocessor-based machine at a time (Condor included).

Thus, we instead take the approach of treating the proxies in the backup pool as part of the active pool. In our implementation, the backup proxies fill in empty slots in the extendible hash table described previously (or extend the size of the has table as needed); ideally, the backup proxies divide the work of those active nodes that are the most loaded (but not yet overloaded enough to require the addition of a new active node). Thus, this approach has the benefit of potentially increasing proxy performance with little effort.

We note that if a backup proxy fails, then no subsequent action is needed. The natural behavior of the extendible hashing algorithm will redirect traffic to the closest parent node. Conversely, if an active proxy fails, then one of the backup proxies must take over its data set; ideally, the backup proxy that is the least loaded should be picked for this change.

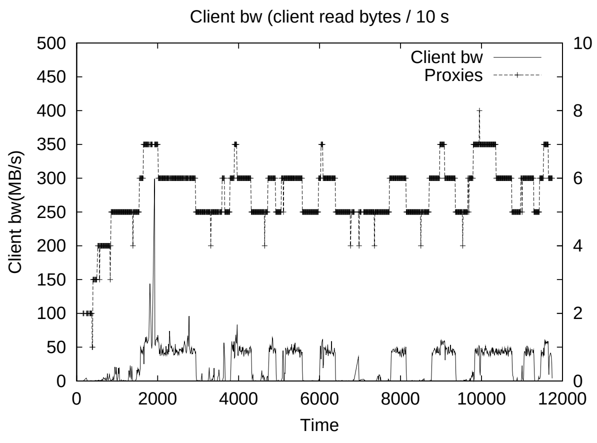
One drawback of treating backup proxies as part of the active pool is that this somewhat complicates the sizing of the active pool. With a backup proxy in place, the Hawk monitor cannot easily observe whether the active proxy (with which the load is split), would be overloaded without the backup proxy. To compensate for this sharing, the Hawk monitor must determine if the sum of the working sets being handled by these two proxies would cause the active proxy to thrash. If so, the Hawk monitor increments the size of the active pool and requests that another proxy be created. Note that in this case, the backup proxy is simply reclassified as an active proxy and a new backup proxy is added to the system; thus, no data is reallocated across proxies, as desired.

4.3 Experiments

4.3.1 Behavior Over Time

In the experiments shown in Figure 3, we investigate the behavior of the system with and without proxies in the backup pool. In both cases, we induce resource failures at random points throughout the run; the exact times at which a failure occurs can be seen by noticing a drop in the number of proxies as a function of the time shown along the x-axis.

In the leftmost graph, we observe performance when there are no proxies in the backup pool. As one can see, due to the high MTTRF of Condor, when the number of proxies drops to below a certain threshold, performance suffers immensely. At these points, the working set of the workload no longer fits into the proxy caches, and thus the proxies begin to thrash,



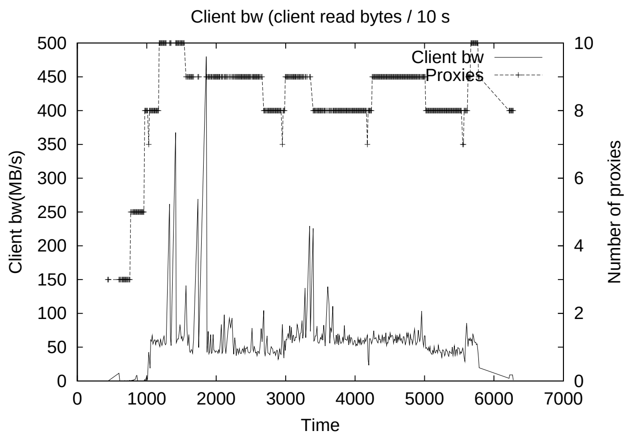


Figure 3: **Sizing For Failure.** *In this experiment, we introduced a high rate of proxy failure to show the effect that maintaining excess capacity can have on throughput. The graph on the left shows the case in which the monitor does not maintain excess capacity. In this case, proxy failures result in a drop in client read bandwidth until the monitor detects the resulting thrashing and returns the needed proxies to the system. With excess capacity as shown on the right, proxy failures have no such negative consequences. Notice the improvement of total runtime when excess capacity is maintained.*

virtually halting all progress. The end result is that the workload runs for almost 12,000 seconds (*i.e.*, over 3 hours).

In the rightmost graph of Figure 3, we show performance when HFS uses its backup-pool strategy. In this case, spare proxies are employed, and when a resource failure is induced, there are enough proxy resources to contain the working set of the workload. Thus, performance is maintained throughout the run, and the overall run-time of the set of jobs is approximately 6,000 seconds, or roughly half the time of the system without any backup proxies.

4.3.2 Varying MTTRF

In our second failure experiment, we explore sensitivity to the accuracy of the MTTRF estimate; specifically, we vary the MTTRF that is induced on the cluster as well as the monitor's estimate of MTTRF. The results of this experiment are shown in Table 1.

From the results in the table, we draw two conclusions. First, having no backups performs substantially worse than a system with some backups, even if the number of backups is smaller than the formula above estimates. Second, having extra nodes in the backup pool (*i.e.*, more that our formula estimates) is often a good idea, given that this affords a margin of safety (especially with correlated failures and recoveries). A corollary benefit of extra proxies in the backup pool occurs because our algorithm for siz-

MTTRF	**None**	**50% under**	**Exact**	**50% over**
300	27717	3919	3770	3440
500	10219	5926	3373	2417
600	15482	4652	3976	3626

Table 1: **Performance Under Varied MTTRF.** *The performance of the backup-pool mechanism is shown under differing MTTRF values. In each experiment, the true MTTRF is set to a particular value (shown in seconds), and the monitor's estimate of MTTRF, instead of being measured dynamically, is set to a particular value, which we then vary across the columns of the table. The column labeled "None" shows the performance when there are no backups in the pool; the column labeled "50% under" shows the performance when the backup pool is half the size it should be; the column labeled "Exact" shows performance when the pool is exactly the size that the formula estimates; fi nally, the column labeled "50% over" shows performance when there are 50% more servers in the backup pool than the formula demands. Performance is reported in the total number of seconds to complete the workload, and the MTTRR for all experiments is set to 200 seconds.*

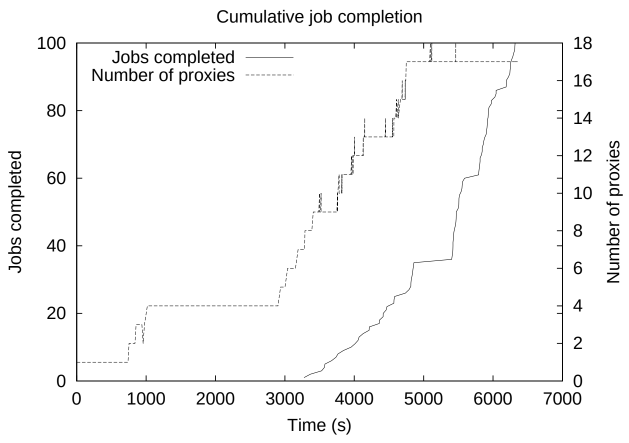


Figure 4: **blastp Performance.** *The figure shows the performance of 100 blastp jobs running on a remote cluster over time.*

ing the active pool is conservative, and thus in some cases the extra proxies increase performance noticeably.

5 Application Experience

Finally, we discuss our initial application experience. We focus on BLAST, a suite of programs used to search protein and DNA databases for sequence similarities [2]. For our measurements, we focus on the blastp program which compares specific protein queries to protein databases. This application reads a small input file (typically fewer than 200 bytes in size) which contains the targeted protein sequence. It then searches for similarities within a large protein database; an example of the type of search performed is shown in Figure 5. Although blastp uses indexing to direct its search, the program will often scan through the entirety of the flat-file database. This database is over 300 MB; together with the index files the blastp program will typically read approximately 350 MB of data during this phase. Following the scan, blastp produces about 16 KB of possible matches.

In the following experiment, we run 100 blastp applications, each searching for a different protein sequence within the shared database. The applications are run upon the same cluster configuration as described earlier. The performance over time is shown in Figure 4.

From the figure, we can make two observations. First, there are two distinct periods of proxy growth. In the first phase of growth (at around 1000 seconds into the experiment), all of the applications are reading the index of the main protein database, and thus the monitor quickly grows the proxy pool to match this demand. The second phase of growth occurs when the first jobs begin to complete; as new jobs then enter the system, re-scanning the protein database, the monitor realizes the value of caching this database, and grows the proxy pool in response.

The second observation we can make relates to the total run-time; the entire workload completes in roughly 6300 seconds. For a comparison, we ran the same workload against a single well-connected network file server, a configuration that is commonly used when manually configuring a cluster to run such a workload. The “local” version ran in roughly 6500 seconds, just a few percent *slower* than our remotely executing version. The benefits of dynamically sizing the proxy pool are clear – if the working set of the application can be captured and spread across a large number of proxies, performance can be better than what one would expect from running in a more standard environment where data is fetched from a single file server.

6 Related Work

A migratory file service shares many of the same goals and challenges of more general systems that support mobile users. One of the primary goals in supporting mobile users is to provide consistent computing services in an ever-changing environment, all transparently to the end user [18]. We share the same goal of transparency. One major difference is that mobile computing tends to focus on wireless environments, a broad range of devices (including sensors and PDAs), and interactive applications, whereas we concentrate on well-connected clusters of workstations running throughput-oriented batch workloads. Thus, our problem is somewhat simpler, since only applications (and not devices) are mobile, the metrics for success are more straightforward (job throughput vs. user satisfaction), and the range of network performance is smaller (no wireless communication). In the future, it may be beneficial to explore where mobile technology for resource discovery could be utilized within our environment.

Our work also shares general issues with that in ubiquitous computing [28]. In the ubiquitous setting,

```
Leghemoglobin   91 IHIQKGVLDP-HFVVVKEALLKTIKEASGDKWSEELSAAWEVAYDGLATAI 140
                   +H  K  +DP +F ++   L+  +    G  ++ EL A+++    G+A A+
Beta globin     91 LHCDKLHVDPENFRLLGNVLVVVLARHPGKDFTPELQASYQKVVAGVANAL 141
```

Figure 5: **blastp Example.** *blastp finds a possible match between broad bean leghemoglobin and horse beta globin.*

most applications assume that its will not change much over time [24]. While this statement was originally made in the context of GUI-based interactive applications, it clearly applies to file servers as well, which traditionally run in highly-managed, relatively-static locales. Thus, one of the main challenges we face in building a migratory file service is to avoid thee static assumptions that are built into most file servers. Schilit *et al.*'s solution is to introduce the concept of a "dynamic" environment variable, which applications use to learn about the characteristics of the current system; such a utility would be useful in our environment as well.

Within the domain of file systems, the Coda project was an early effort in mobile computing that bears similarity to our work [17]. In Coda, users specify in a hoard profile the files they believe are needed for disconnected operation; the system then attempts to balance the demands of the current working set with the users hoard priorities to ensure smooth operation under disconnection. In HFS a migratory proxy could serve a similar role, caching data so as to avoid difficulty when the wide-area link to the home node fails.

Another possible solution to the problem addressed in this paper is to use a wide-area file system or peer-to-peer storage system at each cluster where one wishes to run jobs [15, 27, 22, 5]. Unfortunately, such an approach is often not practical within a Grid, since each participating site must retain autonomy over its own local resources, including the software that is installed. Thus, mandating that all sites run a certain wide-area file system is likely to result in many fewer parties contributing to the global resource pool.

Finally, perhaps the most closely related work to ours is Condor-G, a system that is to Grid computation what a migratory file service is to Grid storage [13]. One of the main mechanisms in Condor-G is the "GlideIn", which enables a "meta" job in a different batch scheduling system to be submitted to the Condor batch scheduling system [19]. Thus, Condor-G allows the user to construct a remote Condor pool to run their jobs, even if the remote site does not have Condor installed. Thus, Condor-G is complementary to HFS. However, Condor-G does not address the problem of data storage or movement beyond the mechanisms built into Condor, which redirect all I/O system calls back to a home node.

7 Conclusions

Storage services are of increasing importance in the Grid environment. In this paper, we present HFS, an instance of a migratory file service, which we believe is a key technology in achieving ease-of-use and high-performance for Grid-based applications. The key to an MFS is the use of migratory proxies and interposition; by redirecting I/O traffic from applications to the proxies, high performance can be delivered; by doing so in a manner that is transparent to applications, ease-of-use is maintained.

Crucial to the success of an MFS is the management of the pool of proxies in the system. HFS manages this pool via a monitor, which observes performance and failure characteristics to dynamically size the pool for maximized performance. Through a series of microbenchmarks and a real application study, we demonstrate the effectiveness of the core HFS mechanisms, showing that they can adapt as desired.

In the future, we plan to better understand the utility of HFS through two specific avenues of research. First, we plan to undertake additional application case studies. Second, we plan to implement our migratory environment within other batch-scheduling systems. The former should assist us in fine-tuning HFS policies, whereas the latter will likely lead us to generalize HFS mechanisms. It is our hope that this combined path will result in a more general and useful migratory file service.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel,

and S. Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 2001.

[2] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, , and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In *Nucleic Acids Research*, pages 3389–3402, 1997.

[3] Avery, P. et al. CMS Virtual Data Requirements. http://kholtman.home.cern.ch/kholtman/tmp/cmsreqsv6.ps, 2001.

[4] Condor. The Condor Directed-Acyclic-Graph Manager (DAGMan). http://www.cs.wisc.edu/condor/dagman/, 2002.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. In *TODS '79*, pages 315–354, 1979.

[7] I. Foster and P. Avery. Petascale Virtual Data Grids for Data Intensive Science. GriPhyn White Paper, 2001.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[10] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *I/O in Parallel and Distributed Systems IOPADS*, pages 14–25, 1997.

[11] I. T. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91, Saint-Malo, France, Oct. 1997.

[13] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, California, August 2001.

[14] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Conference*, June 1998.

[15] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, 6(1), February 1988.

[16] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Dec. 1993.

[17] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1), February 1992.

[18] L. Kleinrock. Nomadic Computing – An Opportunity. *ACM SIGCOMM Computer Communication Review*, 25(1):36–40, January 1995.

[19] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[20] S. Lohr. Supercomputing and Business Move Closer. New York Times Business/Financial Desk, February 19 2002.

[21] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing (HPDC 7)*, Chicago, Illinois, July 1998.

[22] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to- peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[23] M. Satyanarayanan. Digest of the Seventh IEEE Workshop on Hot Topics in Operating Systems. www.cs.rice.edu/Conferences/HotOS/digest/digest-html.html, March 1999.

[24] B. N. Schilit, M. M. Theimer, and B. B. Welch. Customizing Mobile Applications. In *USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, Cambridge, Massachusetts, 1993.

[25] D. Thain and M. Livny. Multiple Bypass: Interposition Agents for Distributed Computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[26] The PBS Implementation Team. The Portable Batch System. http://www.openpbs.org/, 2002.

[27] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing (HPDC 7)*, July 1998.

[28] M. Weiser. The Computer for the 21st Century. *Scientifi c American*, pages 94–104, Sept. 1991.

[29] S. Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, Dec. 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.