1987

Shadow Editing: A Distributed Service for Supercomputer Access

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Jim Griffioen

Rajendra Yavatkar

Report Number:

87-722

SHADOW EDITING: A DISTRIBUTED SERVICE FOR SUPERCOMPUTER ACCESS

Douglas Comer
Jim Griffioen
Rajendra Yavatkar

CSD-TR-722
November 1987

Shadow Editing: A Distributed Service for Supercomputer Access *

Douglas Comer Jim Griffioen
Rajendra Yavatkar

Computer Science Department
Purdue University
West Lafayette, IN 47907

CSD-TR-722

November 18, 1987

Abstract

Long-haul networks are now being used by the scientific community to access resources at the supercomputing centers. There is a need for high-level user services and tools to provide transparent and efficient access to remote resources. This paper describes a way to provide a computational service for supercomputer access in a distributed environment. We discuss the issues involved in the design of such a service and focus on a remote job submission facility. The paper also describes a prototype implementation.

*This work was supported in part by a grant from the National Science Foundation's Office of Advanced Scientific Computing (ASC-851-5851), with additional support from Purdue University.

1 Introduction

Computer networks provide an opportunity to combine geographically dispersed researchers and computing resources into an integrated computing environment. Recently, the National Science Foundation started a networking program (NSFnet) for the benefit of scientists [JLF*86]. The NSFnet will provide scientists access to supercomputers located at various Supercomputer Centers. Unfortunately, adequate user services to access the remote computing sources are not yet available, and there is a need to provide capabilities to the scientific community to make effective use of the interconnected networks in accessing the remote resources [Lei87]. In [Lei87], Barry Leiner points out that the function of remote job entry (an ability to submit batch jobs for processing to remote hosts and receive output) is among the various capabilities that must be supported in an internetworking environment.

Currently, all major supercomputer manufacturers provide the function of remote job entry in their operating systems or front-ends, but that support does not generalize well in the presence of networks. For example, extending a conventional remote job entry (RJE) utility into an internetworking environment does not work well. In a naive implementation, the client must transfer all the files needed for remote processing over the network every time he submits a job. Such an implementation does not use the network resources effectively. Under this scheme, a scientist shoulders the burden of dealing with multiple access procedures, operating systems, and communication details.

In addition, the presence of network connectivity adds new flexibility to the user environment and makes new demands on a RJE utility. In such an environment, RJE should support separation of job execution and output delivery. For example, a capability to submit a job from a host and have its output delivered to another host (with special facilities such as a high-speed printer) would be desirable.

This paper describes an effort to provide a transparent and efficient computational service for supercomputer access in a distributed environment.

The remainder of this paper is organized as follows: In sections 2 and 3, we describe the background and objectives of this project. Section 4 provides a brief overview of the system. In section 5, we discuss the design issues. Sections 6 and 7 describe the prototype design and implementation. The concluding section reports the results of performance measurement and outlines our plans for future research.

2 Background

2.1 Motivation

There are two approaches to providing access to the supercomputers, namely, provide facilities for remote interactive login, or provide a batch facility for remote job entry. Currently, for example, the scientists accessing the supercomputers at a NSF-sponsored supercomputing center use the remote login approach: a user uses a remote login service to start an interactive session, transfers all the files needed from the local environment to the supercomputer environment using a file transfer facility, and then invokes suitable commands on the remote system to execute his job. He then either waits for the completion of the job, or periodically accesses the remote host to determine the status of his job. At the completion of the job, he retrieves the results himself again using a file transfer facility.

Clearly, the remote login method is cumbersome from a user's point of view.

The user must learn the tools and application interfaces for a new environment (the supercomputing site), must concentrate on the extraneous details of file transfer and remote login, and must also poll the remote host for the status of his job. Also, because a supercomputer serves several users, it is likely to be swamped with several such remote login and file transfer sessions. The remote login approach is also inefficient in its use of the network resources.

A remote batch facility for job entry accepts requests for job execution and carries out the transfer of data and results to and from the remote site. A typical scenario under such a batch system for supercomputer access is as follows: A scientist edits files at his workstation using a conventional text editor and submits a request for remote execution on the supercomputer. The request contains a set of commands along with a set of program and data files. Submitting the request to the supercomputer involves transferring all the pertinent files over the network. If the files are large, the amount of time spent in file transfer is significant. At the end of execution, the scientist fetches the results over the network. Typically, the scientist repeats this edit-submit-fetch cycle several times until the programs and data are correct.

Typical long-haul networks currently in use employ low-speed lines (9.6k to 56Kbps) [HHS83,MB87,CNY87]. Repeated submission of a batch job involves transferring the same files again and again even though the files change little. Our goal is to provide an efficient RJE system for accessing supercomputers over low-speed lines. We achieve this goal by taking advantage of the small changes to files between two successive editing sessions.

2.2 Assumptions

The orientation of our work derives from several observations about the existing networks. First, the long-haul networks currently in use employ low-speed lines. As a result, network communication introduces much delay. Transmission delay is significant because scientific computations typically involve large files. The introduction of high-speed lines in the future is not likely to solve the problem because a significant part of user community will continue to prefer low-speed lines for economic reasons. There are other reasons for minimizing the network communication. We expect that the users will be charged for their use of network services in proportion to the volume of traffic generated. More traffic causes the network congestion and results in poor performance [Nag84]. Thus, it is desirable to design a system that minimizes the network traffic and conserves the network resources.

As stated earlier, we assume that scientists repeat the edit-submit-fetch cycle several times until they are satisfied. Submitting a job again often involves transmitting files that have not changed at all as well as others whose edited versions differ from their previous version by a small amount. As we will explain later, this assumption about small changes to files between the editing sessions is important for reducing the network traffic.

We also assume that the details of network communication are unimportant to a scientific user, and hence should be transparent.

3 Objectives

With these basic assumptions, our objective is to design a system for supercomputer access with the following characteristics:

- Transparency: To the greatest extent possible, the user's working environment will remain unchanged. Specifically, any utilities or tools that are currently part of the user's environment should remain usable. There should be no hidden surprises in the output of commands or the state and contents of files. The scientists should be able to invoke remote services by specifying *what* should be done. They should not be required to specify *how* it is done. For example, scientists should not need to learn about new operating systems, network connections, or low-level utility programs that move data from one machine to another. Users should not be required to maintain or set up any state information to ensure the correct functioning of the access system. The system should establish and maintain any such state information automatically without user intervention.

- Efficiency: The main objective of this research is to replace the existing, inefficient methods of supercomputer access with an efficient one. We seek a system that is efficient in using the CPU and network resources. Our goal is to reduce the amount of network traffic, and increase network throughput without significantly increasing the processing overhead at the local workstation or the supercomputer site.

- Customizability: To provide the user with the flexibility to personalize his working environment, we want to parametrize our system to allow easy changes to the environment. For the purpose of transparency, the system should provide a set of default parameters. Users should have their own set of parameters and an option to customize them according to their needs and preferences.

- Adaptability: Because the performance of the system depends on factors that are dynamic (such as the load factor on a system or the network traffic), the system should have a built-in mechanism to tune its performance according to variations in the environment. For example, the system should understand and respond to changes in network delays, cpu speed and load, and file storage capacity. Using this information the system should dynamically tune itself.

- Portability: We desire a system that is independent of specific tools or operating systems so it can be used on a variety of hosts and operating systems and can easily adapt to different environments. Clearly there will be operating system dependencies, but they should be isolated and limited as much as possible.

4 System Overview

As a first step in understanding efficient, high-level computational access services, we have designed a prototype for remote job submission. The prototype provides the user with commands to submit jobs and check the status of outstanding jobs. It retrieves the output at the end of job execution and notifies the user of job completion.

The system operates by setting up a caching store at the supercomputer site. The files submitted as part of a batch job are cached at the remote site (we refer to such files as *shadow* files). The parts of the system are distributed between local and remote site[1]. At the local site, a *shadow* editor encapsulates an existing

[1]In the remainder of this paper, we will refer to the component at the local site as the client and the one at the remote site as the server.

editor (of the user's choice). Whenever a scientist finishes editing a shadow file, the shadow editor notifies the server at the remote site of the change to the file. The server then updates its cached version (if any) for later use. The updates to a shadow file are provided in the form of changes to a previous version. Normally, the changes made are small when compared to the size of the entire file.

5 Design Issues

Several design issues need to be resolved before we can translate our goals into a system design. In the following, we discuss these issues in detail.

5.1 Caching

System designers have used caching effectively in software systems to achieve efficiency [Smi82]. In our system, we use caching for performance improvement by retaining the copies of files submitted at the supercomputer. The basic assumption is that a user is likely to submit the same file or a slightly modified file over and over again. Thus by saving a copy of the file (caching) at the remote host we can greatly reduce network traffic as well as network delays. For example, suppose that a user submits a job and two associated files to a remote host for processing. On receiving the results of the job the user notices that there was a slight error in one of the files submitted. The user corrects the error and resubmits the job. Because the server caches the files on the remote host, the client need not transmit the unmodified file, and the client sends only the changes to the modified file. Clearly the client sends less data across the network that results in reduced network delays for job submission and retrieval.

In addition, caching allows us to introduce some concurrency into the model.

With caching, we can send updates in the background rather than waiting for the user to submit the job again. Again using the scenario described earlier, imagine that both files needed to be modified. After the user modified the first file, the changes could be sent in the background while the user is modifying the second file. It is also possible to send the modifications made to the second file as soon as the user exits the editor and make them effective at the remote host before the user formulates a new job submission request and sends it off.

Another advantage of caching is fault tolerance. That is, caching is a best effort storage system. Caching does not guarantee that a duplicate copy of the user's file will always be available at the remote host. Thus, if for some reason the user's file is lost (perhaps the remote machine ran out of disk space and removed it), the system will still function. The software takes advantage of a cached file if it is at the remote host, but in the worst case it would have to send the entire file. The best effort characteristic of storage offers the system much freedom. That is, it allows us to change some of the parameters to maximize performance. It allows the remote host to decide how much disk space should be used for caching (based on space available, network speeds, disk speeds, CPU speeds, etc) and also which files should be removed from the cache first.

5.2 Data Flow Control

An important component of our system is the data transfer between the local and remote site. A user may not necessarily start the data transfer because the cache may be automatically updated in the background whenever a shadow file is modified. A significant design issue is that of controlling the data transfer. It may either be started and controlled by the client at the local site (we refer to it as *request driven*) or may be controlled by the server at the remote site (referred to as *demand driven*).

The request driven model suffers from the disadvantage that it puts the responsibility of data transfer at the wrong end. Because the server maintains the cache, the server is in a better position to determine when to start data transfer. Also, under this model, the client maintains state information about which files are cached on the remote host and whether the cached versions are updated on a remote host. If the user interacts with multiple remote hosts, it is even more complex to maintain the state and coordinate the updates. Another disadvantage is that if the remote host serves several clients, it may get overrun by such updates disrupting its normal processing and, thus caching may prove counterproductive.

In contrast, under the demand driven model, no state information needs to be maintained at the user site. It gives the server freedom to determine when to update its cache. The flow control at the remote host allows it to take steps to avoid overloading and overruns. The remote host may not run a job immediately. By monitoring the load average, cache size to disk space ratio, number of incoming jobs, network delays, etc., the remote host can decide when is the best time to retrieve the needed files and to schedule and run the jobs. Another advantage is that job submission and update requests are short and quick in the demand driven model because no explicit bulk data transfer is involved.

5.3 Naming

Another design issue we must address is that of naming. There are two aspects of naming that are significant in our system design. A supercomputer normally serves several clients in a heterogeneous environment with different operating systems. Different environments organize their local name spaces in different ways and use different naming conventions. The heterogeneity of the client environments causes the problem of *name resolution*. The first problem concerns a mechanism to resolve a file name at a user site into a globally unique name before presenting it to the server, so the server at the supercomputer site can distinguish among files belonging to different users on different hosts.

The second problem concerns the naming of cached files at the supercomputing site. That site has its own naming space and naming conventions. We need a way to name the cached files so they can be uniquely identified and be located efficiently at the remote site. We postulate that a client at a user site presents a name (a file name local to that site) which is globally unique (unique over all clients).

Given such a unique name and a method to name cached files, we need a mapping function at the remote site that maps a unique file name presented by the client into the name of the corresponding cached file. The problem of finding such a mapping function is relatively simple. The function of mapping can be accomplished by maintaining a file that lists the user-specified names and the corresponding shadow identifiers. Such a method is not efficient in locating a shadow file, but we can use conventional hierarchical directory schemes or indexes to organize the names of cached files so they can be located efficiently.

The problem of resolving local names into globally unique names is especially interesting. It is important to us because it is tied to the problem of cache coherence. The question is how to determine whether a name of a file presented by a user is unique. It is possible that a file has two or more names in the user's file system. We want to avoid keeping more than one cached version of the same file. If there is more than one cached copy of the same file, it not only results in waste of disk space, but may result in inconsistent behavior if one of the cached versions gets updated and the others do not.

One way to solve the problem of name resolution is to take a file name at the user site and reduce it to its *basic* file name. We can obtain a basic name by replacing any aliases (symbolic names or links) with the actual name. For instance, in the hierarchical directory systems, a file name specifies the path through the directory tree from its root to the file. Such names are referred to as *full path names*. A full path name can be specified so that the name is unique within a single system. However, to guarantee uniqueness of such names across multiple hosts, one must either include all the client hosts in one directory tree [WPE*83] or prefix full path names with unique host names [BMR82,TR84].

Unfortunately, neither of the above solutions is sufficient to guarantee uniqueness in the presence of distributed or network file systems. A unique file name on a host is not sufficient if the file system crosses machine boundaries. For example, under Network File System (NFS) [SGK*85], machine A may *export* its file system so that other machines can easily access machine A's file system. Likewise machine A may mount file systems being exported by other machines. Thus a file accessed on a given machine may not necessarily belong to a file system located on it. For instance, if machine C is exporting its */usr* file system and machine A mounts it as */proj1* in its directory hierarchy and machine B mounts it as */others*, then both machines A and B would have access to the same file (say */usr/foo* on machine C) however they would both have different names for it (*/proj1/foo* and */others/foo*).

In the Tilde file system [CM86], the names used by a user need not be globally known or globally unique. Tilde scheme organizes the directory system into a set of logically independent directory trees called tilde trees. Files within a tree are accessed using the tree's tilde name and a pathname within that tree. Each user specifies his own tilde trees that reflects his personal view of the hierarchy in the file system. The actual location of the files is of no consequence to the user and the files may migrate from a machine to another without altering the user's view. Different users may refer to the same file by different tilde names. An absolute name is associated with each tilde tree and is unique across all machines. But a set of absolute tree names is associated with each user and a user may occasionally change the set of absolute names. An absolute name alone is not sufficient to uniquely identify a file.

With the diversity of naming systems available, clearly there is no single naming scheme that can be used to resolve names under different schemes. Our approach is to view the client's name space as consisting of a domain and a unique file name within that domain. For example, a domain may span a single host or a collection of hosts as in a NFS environment or in the Locus distributed file system [WPE*83]. We expect a mapping function at the local site to localize the details of the naming scheme used under that domain. That function maps each local file name into a (domain id, unique file id) pair and presents it to the remote site. We assume that each domain can be identified uniquely on a global basis (for example, an internet network number may serve as a unique domain id in the internet world).

Given such an organization on a global basis, our system divides its name space at the remote site into domains. For each domain, it maintains a directory that maps each file identifier within that domain into the unique identifier of the cached version.

6 Prototype Design

6.1 System Model

Our prototype is designed to provide users with a remote job submission facility. Under our model, a user edits files at his workstation and uses a batch subsystem to submit jobs for remote execution. The system is responsible for retrieving the results of execution and notifying the user of job completion.

The batch subsystem is based on a client-server model. A *shadow server* runs at each supercomputer site. The client part of the system resides and runs at the user's workstation. The interactions between the client and the server are completely transparent to the user. The server accepts requests for job execution, initiates execution at the supercomputer, reports on the status of outstanding jobs, and transfers results back to an appropriate client. The client hides the details of communication, and accepts requests for remote processing at the user's site. Multiple clients can have connections open to a server simultaneously, and a client can have simultaneous connections to multiple servers.

6.2 User Interface

A set of commands to edit files, submit jobs for remote execution, and to check the status of the incomplete jobs constitute the user interface. In addition, a *shadow environment* stores the information about each user. Though the environment is set up automatically, a user has an option to customize it according to his own choice. Currently the prototype supports the following operations:

- Shadow Editor: Shadow Editor encapsulates a conventional editor of the user's choice (specified through an environment variable). It does not modify an existing editor and the user's view of the editor remains unchanged. It contains a postprocessor responsible for carrying out tasks related to shadow processing at the end of an editing session.

- Submit: The submit command accepts a list of file names, the name of a job command file and a few optional arguments. The job command file contains one or more lines where each line specifies a command (along with its arguments) to be executed at the remote host. The list of files supplied with the submit command specifies the data files needed for processing these commands. The submit command returns a job identifier that can be used subsequently to query the status of the job. After a job is executed, the output and the errors (if any) are returned automatically. The optional arguments allow the user to specify the names of files into which the system stores output and error messages. Because a user may access more than one supercomputer, the hostname can be specified as an optional argument. If no hostname is specified, the system submits the job to a default host specified in the shadow environment.

- Status: The status command, which accepts a job identifier as an argument, allows a user to find out the status of a job submitted earlier. If the job identifier is not specified, status returns the status of all the jobs pending at the remote host. The client contacts the shadow server to obtain the current status of an incomplete job. When remote execution of a job completes, the shadow server contacts the client to transfer the output. The client maintains the information on the status of all the jobs.

6.3 Customization

6.3.1 Shadow Environment

The shadow environment is a database that contains the information about the status of all the jobs submitted and customization information for each user. It also contains the information needed for managing the different versions of a file (*version control*). Because the shadow processing involves both client and server sites, the parts of the shadow environment are distributed across local and remote sites.

6.3.2 Version Control

As part of the shadow environment at the remote site, we cache a copy of each file submitted by a user. The system keeps track of the different versions of the same file stored at the remote and user site. When a user submits the same file again, the system computes the difference between the previous version at the remote site and the current version, and transmits the changes instead of the entire file.

On the client side, the system associates a version number with each file. Thus, every time a file is edited, a new version is created and identified separately from the previous versions. When the shadow server requests a file, it indicates which version it has along with the file name. In response to such a request, the client may transmit a completely new version (if the specified version is not available for computing the differences), or the difference between the current version and the previous version specified by the server. In the latter case, the server computes the correct version using the old version and the set of changes received.

As the user repeatedly edits a file, several versions of that file are created at the client site. To avoid retaining the old versions indefinitely, the client deletes older versions after the server acknowledges the receipt of a later version. In addition, a user may specify, as part of customization, a limit on the number of older versions that should be retained at any time.

6.4 Client-Server Interactions

In view of the caching of data files at the server site, the key aspect of the client-server interaction is maintaining the coherency of the server cache. Keeping the cached versions up-to-date helps in minimizing the amount of data transferred. We use the demand driven flow of control in the exchange of information between the client and the server. The client informs the server whenever it creates a new version of a file. The server may retrieve the changes to a file immediately after it is notified of a change, or it may postpone such a retrieval until the changes are actually needed. Thus, the server has the flexibility to control the information flow depending on the availability of its resources.

A typical scenario is as follows. When a user finishes editing a file, the client contacts the server to notify it about the creation of a new version. The server, in turn, may request the client to supply the updates immediately , or may postpone such a retrieval for a later time. In response to a submit request from a user, the client contacts the server and supplies it with the job control file, the names of data files and their version numbers. Depending on the system state, the server may process such a request immediately or queue it up for later processing. The updates for the files involved may be obtained in the background even before a submit request is received and processed.

6.5 Name Resolution

As described earlier, we resolve the file names at a local site to a unique (domain id, file id) pair for the domain that a site belongs to. In the simplest case, a site may constitute a domain, or a collection of sites as in [WPE*83].

Our prototype works in a UNIX/NFS environment and uses the following method. We use an algorithm to resolve a file name at a local site using the NFS specific semantics. Our algorithm resolves aliases, symbolic links and retrieves a unique absolute path name for the file within the local host. If any prefix of the path name belongs to a mounted file system, it then consults the NFS mount table to resolve that prefix further on the host that exported that mounted file system. It iterates through the resolution process (there are no circularities allowed in NFS) until a file name is resolved to a unique (host id, path name) pair within the NFS domain.

Once a unique file identifier is obtained for the local domain (which consists of a set of NFS sites), the remote site maintains a separate mapping file for each domain that maps each file identifier within that domain into the name of the cached file at the remote site. Even if a user submits the same file from two different hosts within a NFS domain, there will be a single cached copy of that file at the remote site.

7 Prototype Implementation

Our current prototype is implemented under the UNIX[2] operating system (4.3 BSD and SUN 3.3). The environment consists of several UNIX workstations and mainframes that together run the Network File System (NFS) [SGK*85]. A remote UNIX system currently serves as the supercomputer. Clients and servers are implemented as UNIX processes that use a reliable transport protocol (TCP/IP) for interprocess communication. A server process listens at a well-known port for connections from clients. A client process runs at each user workstation and is responsible for interaction with the server. We use the Cypress network [CNY87] as the underlying network for communication between server and clients. Cypress provides low-cost technology for Internet access and is suitable for setting up capillary connections from user sites to the NSFnet backbone.

To compute the changes to a file after editing, we use an algorithm for differential comparison [HM75] (available under Unix as the *diff* command). The algorithm computes changes in a form suitable for an editor (like *ed* in Unix) to apply the changes to a previous version to update it to the later version. We have conducted a number of experiments to evaluate the performance of our prototype, the results of which are presented in the next section.

8 Conclusion

8.1 Performance

We chose two existing networks to measure the performance of our system. The Cypress network uses 9600 baud lines and has been in use for over a year. Cypress

[2]UNIX is a registered trademark of AT&T Bell Laboratories.

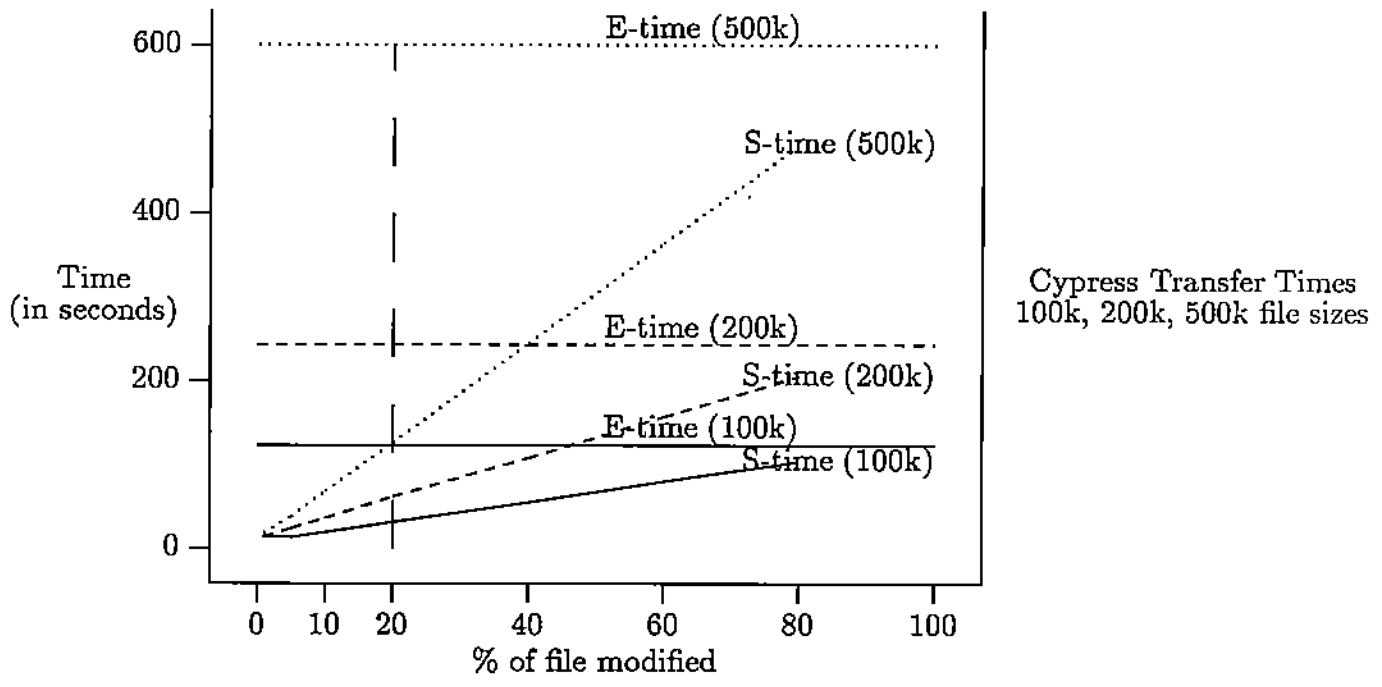
provides a low-speed, low-cost technology for accessing the Internet and is useful for providing capillary connections to NSFnet. We chose ARPANET for our experiments because it provides an example of a wide area network with high-speed connections (56K bps). ARPANET serves several sites and handles a large amount of traffic as will be expected when remote access to supercomputers using a long-haul network becomes common.

We used files of different sizes (ranging from 10K to 500K bytes) in our experiments. In each experiment, we submitted a job with a data file. After obtaining the results, we edited the data file and resubmitted the same job. We modified the data file by a different amount every time (the amount of text modified varied from 1 of the text to 80 of the text) before resubmitting the same file. We measured the total amount of time spent in each case.

Figure 1 shows the results for Cypress. A horizontal line shows the amount of time needed when a file is submitted for the first time (which involves transferring the entire file) and hence corresponds to the time needed under a conventional batch system.

For experiments using ARPANET, we chose a supercomputing site close to Purdue (the University of Illinois). Because our software is still at an experimental stage, we cannot install it at a production site such as Illinois. Therefore, we estimated the times for file transfers under our system by measuring the times using FTP [PR85]. Figures 2 and 3 show the results for ARPANET.

The results show a dramatic improvement in the performance using shadow processing as opposed to conventional batch processing. If we assume that an editing session modifies less than twenty percent of the original file, then our plots show that the entire processing is four times faster under our system than



E-time (500k)
S-time (500k)
E-time (200k)
S-time (200k)
E-time (100k)
S-time (100k)
600
400
200
0
Time
(in seconds)
0
10
20
40
60
80
100
% of file modified
Cypress Transfer Times
100k, 200k, 500k file sizes

Figure 1:

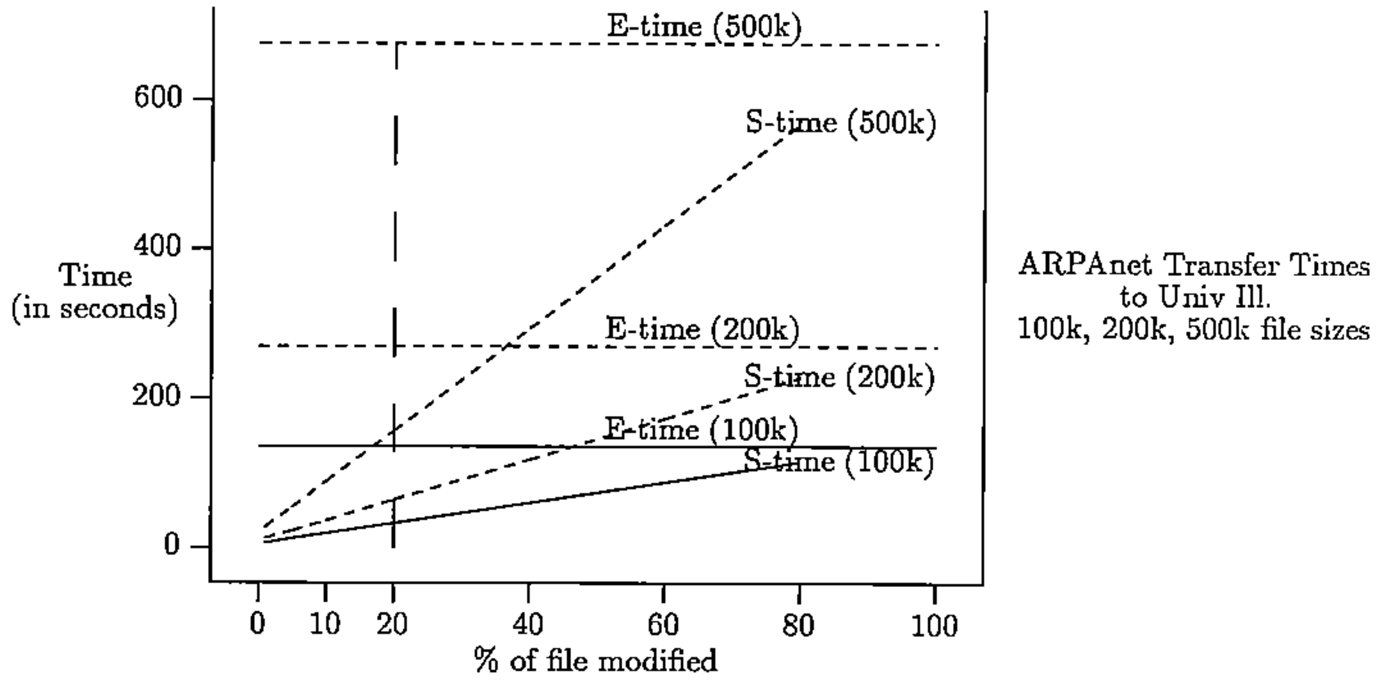


Figure 2:

File Size	Speedup Factor*			
	1% modified†	5% modified	10% modified	20% modified
10k	13.5	9.3	6.5	3.7
50k	22.5	11.9	7.1	4.3
100k	24.2	12.0	7.5	4.3
500k	24.9	12.5	7.6	4.3

* Speedup Factor is $\frac{E-time}{S-time}$. (Data gathered from ARPANET)

†Percentage (in bytes) of text that was modified

Figure 3:

that under a conventional batch system. In practice, when large files are involved (around 100K bytes or more), it is likely that a user will modify less than five percent of the entire file. In that case, our system is up to twenty times faster than a conventional system. The plots also show that the improvement in performance is significant even if a large portion of a file gets modified.

The results obtained with ARPANET (which uses 56 Kbps lines) are significant because they show that the utility of our system is not limited to networks using low-speed lines. Even if we use higher speed lines (56 Kbps and higher) in a backbone network (such as NSFnet), the effective bandwidth available to individual users will be less due to the large number of users and congestion problems as evident in ARPANET [Nag84].

8.2 Summary

In this paper, we have described a system for providing transparent access to supercomputer services. The performance evaluation of the prototype demonstrates that shadow processing achieves significant speed-up in remote job execution. The results are significant even for a long-haul network like ARPANET that compares well with the NSFnet backbone.

8.3 Future Work

We are pursuing the protocol development to add more functionality. For example, we plan to provide additional services such as routing the output to different hosts. We plan to investigate the problem of name resolution further to arrive at a general solution.

Sometimes the result of processing on a supercomputer involves generating a large amount of output (For example, graphics applications). In such a case, it will be advantageous to apply the technique of shadow processing in reverse (i.e., cache the output on supercomputer, and, next time the same job is run, send the differences between the current output and the previous output to the client). We need to examine this option further.

There are different algorithms proposed to compute the differences between two files [MM85,Tic84]. We will study these algorithms and adopt the one that offers better performance. We also plan to explore data compression techniques to improve the efficiency of data transfer. Finally, we believe that the best way to evaluate our system is to use it. Therefore, we plan to make it available to the scientists to access the supercomputer services at Purdue.

References

[BMR82] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle connection or UNIXes of the world unite! *Software–Practice and Experience*, 12(12):1147–1162, December 1982.

[CM86] Douglas Comer and Thomas P. Murtagh. The Tilde file naming scheme. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 509–514, May 1986.

[CNY87] D. Comer, T. Narten, and R. Yavatkar. *The Cypress Network: A Low-Cost Internet Connection Technology.* Technical Report CSD-TR-653, Computer Science Department, Purdue University, April 1987.

[HHS83] R. Hinden, J. Haverty, and A. Sheltzer. The darpa internet: interconecting heterogeneous computer networks with gateways. *Computer*, 16:38–48, 1983.

[HM75] J.W. Hunt and M.D. McIlroy. *An Algorithm for Differential File Comparison.* Technical Report Computing Science Technical Report 41, Bell Laboratories, 1975.

[JLF*86] D.M. Jennings, L.H. Landweber, I.H. Fuchs, D.J. Farber, , and W.R. Adrion. Computer networks for scientists. *Science*, 231(28):943–950, 1986.

[Lei87] Barry Leiner. Network requirements for scientific research. Internet Task Force on Scientific Computing, August 1987. RFC 1017.

[MB87] D. Mills and H-W Braun. The nsfnet backbone network. In *Proceedings of the ACM SIGCOMM Workshop on Frontiers in Communication Technology*, AUGUST 1987.

[MM85] W. Miller and E.W. Myers. A file comparison program. *Software–Practice and Experience*, 15(11):1025–1040, November 1985.

[Nag84] J. Nagle. Congestion control in tcp/ip networks. ARPANET Working Group Requests For Comments, January 1984. RFC 896.

[PR85] J. Postel and J. Reynolds. File transfer protocol. ARPANET Working Group Requests For Comments, October 1985. RFC 959.

[SGK*85] R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network file system. In *Proceedings of the Summer USENIX Conference*, USENIX Association, June 1985.

[Smi82] A. Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.

[Tic84] W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[TR84] Walter F. Tichy and Zuwang Ruan. Towards a distributed file system. In *Proceedings of the Summer USENIX Conference*, pages 87–97, USENIX Association, June 1984.

[WPE*83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.