Lerna: An Active Storage Framework for Flexible Data Access and Management

Stergios V. Anastasiadis* Rajiv G. Wickremesinghe Jeffrey S. Chase
Department of Computer Science
Duke University
Durham, NC 27708, USA
{stergios, rajiv, chase} @ cs.duke.edu

Abstract

In the present paper, we examine the problem of supporting application-specific computation within a network file server. Our objectives are (i) to introduce an easy to use yet powerful architecture for executing both custom-developed and legacy applications close to the stored data, (ii) to investigate the performance improvement that we get from data proximity in I/O-intensive processing, and (iii) to exploit the I/O-traffic information available within the file server for more effective resource management. One main difference from previous active storage research is our emphasis on the expressive power and usability of the network server interface. We describe an extensible active storage framework that we built in order to demonstrate the feasibility of the proposed system design. We show that accessing large datasets over a wide-area network through a regular file system can penalize the system performance, unless application computation is moved close to the stored data. Our conclusions are substantiated through experimentation with a popular multi-layer map warehouse application.

1 Introduction

Recently deployed high-speed connectivity across wide-area networks encourages collaboration and resource sharing among autonomous groups at an unprecedented scale. Scientific and commercial data are undoubtedly recognized as invaluable resources, and several new systems have been proposed to facilitate wide-area information sharing in a secure, flexible and cost-effective way [5, 6, 9, 12]. Since the price of network bandwidth remains significant and latencies due to the speed of light make round-trip delays unavoidable, it makes sense to move data processing close to the storage servers.

In order to address these issues, researchers usually do on-demand replication of entire files locally to the sites where computational resources exist to run the applications [8]. But entire file downloading remains non-practical in several cases due to the excessive local storage space required, the long delays involved in bulky data transfers, or consistency issues arising when creating multiple file replicas. Alternatively, there are studies advocating to run application code within the storage devices themselves, capitalizing on technological advances in processing power and memory density [22]. This is not likely to become practical very soon, either, due to the challenges involved in enhancing the currently simple disk interface to allow execution of application code in a general, safe and inexpensive way. Finally, middleware solutions have been developed that dynamically transform data close to the storage server [9, 12]. Even though such functionality is available today, existing systems usually require customization of the applications that they provide support for.

In the past, data communication through files has served as a convenient medium for applications to receive input and generate output. Distributed file systems have generalized this model across different hosts without the need to change the file system interface. Nevertheless, transfer of control to local or remote computation usually follows its own separate path through local or remote procedure calls. The picture is blurred in database systems and web servers that dynamically interpret the data requests and return the selected content to the user. However, dynamic interpretation of data requests is currently only reachable through specialized access interfaces or query languages, and applications should be custom-developed in order to take advantage of them.

In this paper, we verify that accessing large datasets over a wide-area network through existing file systems limits the application performance due to round-trip delays and network protocol processing involved in individual block

*Current affiliation of S. V. Anastasiadis: Department of Electronic and Computer Engineering, Technical University of Crete, Chania 73100, Greece and Institute of Computer Science, Foundation of Research and Technology Hellas, Heraklion 71110, Greece.

transfers. Then, we demonstrate how to functionally and semantically enrich the file system interface to support invocation of application code as a side-effect of I/O requests. We significantly improve the data access throughput, and increase the overall system performance. Thus we show that any application communicating through files can make use of distributed services running as plugins of a network file server. Additionally, the network file server provides the appropriate semantic level to intercept and meaningfully analyze the system I/O traffic. Such information can offer important insight about the usage pattern of the server resources and help improving their management.

The remainder of this paper is organized as follows. In Section 2 we provide additional motivation for the functionality that we introduce, in Section 3 we get into details about the architecture that we propose, and in Section 4 we explain our system prototype and the operation of a multi-layer map warehouse. In Section 5 we go over the experimentation environment that we used, and present our experimental results. Finally, in Section 6 and 7 we summarize previous related work and outline our conclusions.

2 Motivation

From previous research, we already know that enabling processing close to the stored data reduces the amount of network bandwidth and client storage space required within a distributed system [3, 9, 16, 22]. Published literature and anecdotal evidence also supports the argument that building non-trivial I/O-intensive applications requires a significant amount of programming effort for transforming and reading data [10, 12]. Usually the application user does not control the specific format in which data are becoming available. This makes necessary to run commodity or custom-built software in order to provide a desirable data view to the application. If the data is made available at a remote site, extra effort may be needed to manually copy the data locally, and keep the local replica up-to-date. In summary, distributed computation requires software that provides flow-control, filtering and aggregation functionality between the data-producing source and the data-consuming application.

In order to solve the problems involved in remotely accessing heterogeneous data, several solutions have been proposed in the past:

- The database community advocates placing mediator software close to the data storage to unify accesses through some relational, or object-oriented data model [12, 20].
- The supercomputing community has developed approaches that automate the replication of entire files between sites that produce and consume the data [5, 8].
- The systems community previously described object-oriented file systems that attach data transformation methods to the stored data, and migrate the execution close to the client or server depending on the run-time conditions [3, 26].

Despite the elegance of these approaches, none of them is suitable for solving the general data transformation problem. System designers are still more comfortable with developing customized solutions for individual applications. Reasons include preference towards using a particular programming environment, interface limitations of the applications involved, software availability on specific hardware platforms, and performance degradation as a result of generic data transformation software. Under such circumstances, we believe that application development would be facilitated considerably, if the underlying system provided the necessary framework to allow (i) efficient execution of computation close to the stored data, (ii) direct access to the generated data by both legacy and custom-built applications, and (iii) resource utilization tracking for informed data reorganization.

Indeed, we propose to make available within the network file server an execution environment that will allow both system administrators and regular file users to run executable code. Even though previous research has independently proposed file system extensibility [24] and application computation within a storage system [3, 22], we attempt to unify the previous efforts into a more general facility that supports both these types of computation. We strive to achieve our objective by hiding the data transformation operators behind regular file access requests. When an access request to a particular file name fails, the server can intercept the failure, decode appropriately the name, and produce on-the-fly the requested content. A privileged user can register methods in advance with the server to recognize specific types of access failures. Additionally, for every request we can record its type and the time it takes to complete. The approach that we propose is (i) intuitively straightforward because data transformation operators are triggered by usual file accesses, (ii) efficient due to the software maturity of the file system access paths, and (iii) extensible because failure interception modules can be added dynamically to the system.

Despite the similarity of the functionality that we propose to that of dynamic-content web servers, there is a main difference between the two that arises from the inherent interface simplicity of a network file server. This makes *Lerna* appear as a local filesystem to the applications. As a result user authentication and access control are simplified, while data transfers can be done efficiently. It is exactly this advantage of the file servers that we are preserving in our approach so that we can allow the transformation and transfer of remote data to be both cost-effective for the applications

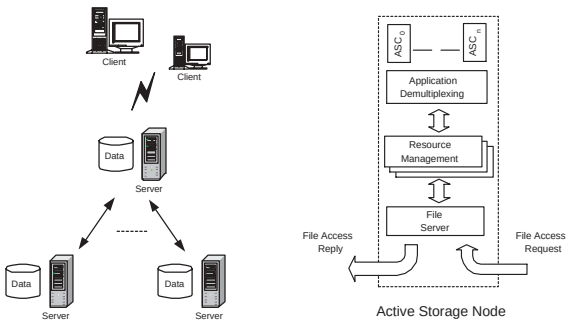

Figure 1. (a) An active storage system consists of front-end and back-end nodes. Each node combines data storage functionality with application-level computation and system resource management. (b) Each node of the *Lerna* active file server consists of application-specific computation and resource management modules running on top of a local file system.

and backwards compatible with legacy software.

Nevertheless, several questions remain open, and it is our goal to clarify them. The main contributions of the present paper include: (i) design and prototype implementation of an interface architecture that allows integration of applications into the file server, (ii) performance measurements of real applications that demonstrate the advantages of *Lerna* in comparison to alternative remote data accesses, and (iii) development of a statistics gathering facility for accumulating I/O traffic pattern information within the file server, which we use to balance the access load across multiple storage devices.

3 The *Lerna* Architecture

The *Lerna* active storage system combines multiple processing/storage server nodes into a multi-tier hierarchy (Figure 1(a)). A server node can access data directly from its storage devices, and also run application-specific tasks for data filtering and transformation. In addition, it can issue remote file system requests to access application services running on a different server node. This provides the framework for potentially decomposing an application service into multiple stages each handled by a different server node. We assume that all the server nodes have the same multi-layer structure. Each request received through the interface of a remotely mounted file system is passed to a vertical stack of resource management layers. The top layer forwards the request to one of the registered modules for application-specific computation (ASC) (Figure 1(b)). In the rest of this section, we describe in more detail the architecture of *Lerna*, and explain the mechanism that we use to access remote application services through the regular file system interface.

3.1 Remote Service Invocation

Clients mount the active storage server as a remote file system (Figure 2). Access of existing files and directories appears to the client as a normal file system request. In addition, every request can potentially be intercepted by a *Lerna* resource management layer for resource accounting or scheduling. Access to an inexistent file triggers execution of a registered ASC module. The name of a requested file contains the identifier and the parameters of the requested service invocation. It is possible to use filename encoding schemes similar to naming conventions of either virtual directories [13], or uniform resource locators for dynamic web pages. In our prototype, we use the URL-encoding format [7] to transform arbitrary command-line statements into permissible file name strings.

When a name LOOKUP fails, the demultiplexing layer of *Lerna* decodes the name of the missing file to the identifier and the parameters of the implicitly requested service (Figure 1(b)). Then, the missing file is created according to the access permissions of the requesting user. Its file descriptor becomes the standard output file descriptor of a new process that is spawned for running the requested service. We save the parameters of the failed LOOKUP call into the data members of a new object that we instantiate. We use a hash table to associate the reference of the new object with the identifier of the service process. This is equivalent to a callback registration for the failed request. In the meantime, the LOOKUP call of the client application remains blocked.

When the service process exits, we use the registered callback to resubmit the original LOOKUP request to the local file system. The new LOOKUP request successfully returns a file handle that is passed back to the file system client. The original call of the client application returns as if the accessed file was there before the request. At this point, the client can proceed with a file READ request, and the file system server returns the requested data block. This is exactly the control sequence that we actually implemented for initiating application-specific computation on the file server. In our prototype, we left file writing unmodified so that written data are passed verbatim to the specified file. In our execution model, a remote service can receive data input from the client through files specified as parameters during its invocation. The invoking client places the input files in the file system already mounted from the server. When the remote service is called, it stores the output into the file that the client attempted to read.

For instance, we make available the `/path` directory locally to the *Lerna* server, and remotely to the *Lerna* client. We store a map configuration description in the text file

`/path/foo.map`, and use the binary `/usr/bin/shp2img` to translate the map description into a map image. Assuming that `/path` is the current working directory of the client, when attempting to execute the command-line:

```
/bin/ls %2Fusr%2Fbin%2Fshp2img+-m+%2Fpath%2Ffoo.map
```

and the file:

```
%2Fusr%2Fbin%2Fshp2img+-m+%2Fpath%2Ffoo.map
```

does not exist, the *Lerna* server will URL-decode the filename into:

```
/usr/bin/shp2img -m /path/foo.map
```

and execute it. The computation output is subsequently stored into the file:

```
/path/%2Fusr%2Fbin%2Fshp2img+-m+%2Fpath%2Ffoo.map
```

and the previous `ls` command returns as usual.

The above description easily extends to cover communication of control and data within a multi-tier server organization, where computation initiated by a front-end node requests files generated by back-end nodes. Currently, a READ access returns control to the client when the entire requested file becomes available. Additionally, our model can also handle the case where file blocks are generated in a streaming fashion as a result of sequential access requests from the client.

3.2 Namespace Structure

Ideally, the remote service and the client should have the same view of the file name space in order to effectively communicate via file exchange as described previously. For example, when an input file is passed from the client, the remote service should be able to find the file in the directory structure visible within the file server. Similarly, when an inexistent file has to be created, the remote service needs to create the file under the directory from which the client requested it. In practice, there is the file server that lies between the client and the application service. The server itself communicates with the client through file handles instead of entire directory paths. It also allows the application to access data directly from the local file system through local file names. Therefore when a new file has to be created under a particular directory, the file server needs to reconstruct and pass to the local application service the entire directory path. In other words, we need a lookup structure that translates directory handles into paths.

One way to achieve that is to maintain within the file server a data structure that keeps track of the hierarchy of individual path components as they are requested by the client. Every time a new edge of the directory tree is successfully looked up, a new record can be created that associates the file handle with its name and a pointer to the record of its parent. Thus, it is possible to gradually build a data structure that we can use to look up file handles and return file path names. Then, when a new file has to be created under a specific directory, we lookup the directory handle in the data structure, and rebuild on the fly the entire path of the file, which we can pass as a string to the application service. An alternative approach that is simpler to support but less intuitive to use would require from the client to pass the path of the current working directory as an additional parameter to the remote service.

3.3 Resource Management

Lerna provides the necessary framework to keep track of the number and completion time of individual block READ and WRITE requests handled by a server node. The total number of READ and WRITE requests along with their average completion time are reported for each individual accessed file. For each file system we use a separate hash table to accumulate usage statistics for its files that have been accessed. We demonstrate in Section 5.6 how we can use this information to find out which files are responsible for most of the system load, and reorganize them accordingly across the different storage devices to improve the system throughput. We should point out that intercepting any accessed block can incur significant computational overhead. Therefore, we normally expect that application software running within the server will bypass the accounting overhead, and access the data directly from the local file system in the common case.

The utilization information accumulated in resource management modules can also be used to make scheduling decisions about the rate of forwarding requests to other parts of the system. Such forwarding can be based on the identifier of the requesting user or the particular application being invoked according to specific quality-of-service policies. Specialized block replacement policies can be used in buffer management according to particular workloads that the server is expected to handle. Blocking or asynchronous semantics can be offered to the clients for accessing files and application services with a desirable degree of flexibility and efficiency.

4 Implementation

In this section, we describe our prototype implementation, and the application that we used to demonstrate and evaluate the system.

4.1 System Prototype

File systems have access to important information about user credentials that can use to offer basic security against

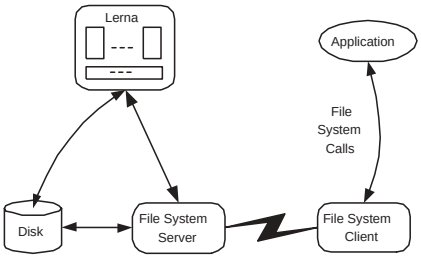

Figure 2. File access requests are intercepted on the server side either for resource accounting, or for dynamic creation of missing files.

Data Layer Description	Format	Size (KB)
States	Shapefile	6,750
Cities and Towns	Shapefile	8,832
County Boundaries	Shapefile	13,346
Hydrologic Unit Boundaries	Shapefile	20,132
Roads	Shapefile	22,624
Federal Lands/Indian Reserv	Shapefile	56,288
Streams and Waterbodies	Shapefile	46,908
Shaded Relief of N.America	GeoTIFF	73,794
Public Land Survey System	Shapefile	74,448
Total		323,122

Table 1. These are datasets containing U.S. geographical information, which are made publicly available by the U.S. Geological Survey (Reston, VA). We distribute the data sets across four separate disks. The shaded relief that is part of every requested map is available across all the disks.

unauthorized access to both data and application computation. Furthermore, the file system interface retains important semantical information about both files and storage devices, which is useful for reorganizing data files according to specific access patterns. Traditionally, file systems were built as part of the kernel in order to reduce data copying and context switching. However, recent research has shown the feasibility of implementing file systems at user space for lightweight access to network storage and improved application control over data movement, caching and prefetching [17, 19]. Since user-level file systems run outside the kernel, they also make straightforward the control transfer to application code.

One example of a user-level network file system is the SFS toolkit [19]. SFS uses a user-level library to provide secure asynchronous communication through the Network File System (NFS). Users submit file access requests to their local NFS client which typically runs within the kernel. The NFS client passes the requests to a user-level client daemon which is part of SFS. The SFS client translates the request to its own format and then passes it through the network to a user-level SFS daemon running on the server. The SFS server daemon translates the request back to NFS format and passes it to the NFS server of the same node. The NFS server transforms the received request to access of the local file system.

In this paper, we describe our experience from offering application- specific computation (ASC) and resource usage accounting with extensions that we made to the SFS toolkit. One advantage of SFS is the self-certifying hostnames that contain server certificates which allow the client to verify the authenticity of the server. By taking advantage of the modular architecture and the facilities of SFS, we needed about 1,000 lines of code to build *Lerna* as part of the server daemon. Our implementation mainly consists of the functionality for demultiplexing failed LOOKUPs into ASC invocations, the implementation of callback registrations, and the usage statistics gathering.

The client of *Lerna* uses SFS to pass input files to the ASC and receive back the computation output. However, the ASC accesses the bulk of the application datasets directly from the local file system of the server node. Thus, we reduce the SFS protocol translation overhead to the minimum necessary. Even though our current prototype is limited to modifying the behavior of READ accesses, and measuring the number and server-side delay of READ and WRITE requests, we believe that other operations can be intercepted and enhanced in similar ways.

4.2 Mapserver

Online generation of geographical, astronomical or biomedical maps generally can consume a large amount of processing and bandwidth resources depending on the size of the datasets involved [9, 25]. In the past, the problem of handling terabyte-sized datasets has been kept manageable by (i) limiting the accessible number of layers to one, and (ii) producing raster images at all the supported resolutions offline so that map rendering could be reduced to data retrieval. In that sense, large-scale management of multi-layer map warehouses remains an interesting open problem with significant implications to both basic scientific research and everyday life. Offline preparation of multi-resolution images solves only partially the problem, due to the substantial online processing required by frequently changing datasets, or multiple layers dynamically selected.

Mapserver is an open-source application developed in a NASA-sponsored project to provide configurable web access to geographical data in the form of map images [18]. Although not a fully-fledged geographical information system, it has sufficient functionality to integrate several different types of data that include raster or vector images, and relational tables from commercial database systems. The Mapserver library can transform map configuration text

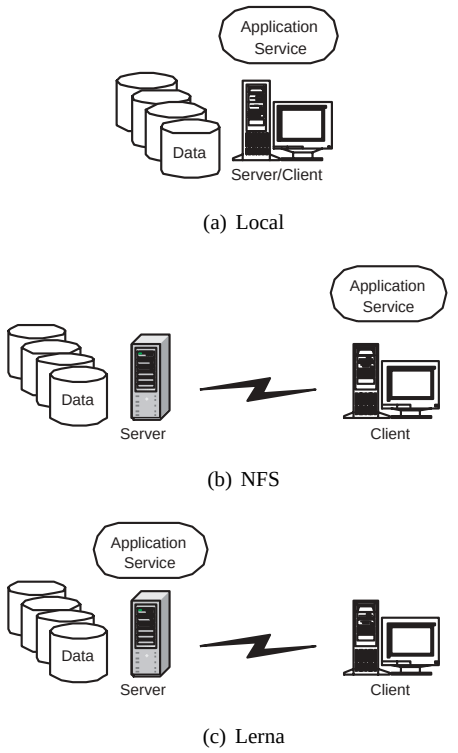

Figure 3. (a) Local computation combines the data storage and processing functions on the same node. (b) NFS clients receive data from a remote server and do the application computation locally. (c) *Lerna* client requests generate data dynamically through triggering application-specific computation on the storage server.

files into raster images that render the specified data layers within a particular geographical region. The configuration files can be generated either manually or through a graphical user interface.

Mapserver scans sequentially the files corresponding to each data layer. The geometrical objects that fall within a specified geographical region are selected and rendered in an internal raster image format. The objects from each layer are superimposed on the internal raster image according to the order with which the layers appear in the configuration file. When the objects rendering is complete, the image is transformed into a supported format specified by the user. We use the Mapserver as a prototype out-of-core application to study the performance advantage from processing proximity to the data, and to investigate the application and system structure implications of invoking computation through file accesses. Our experimentation is limited to servers consisting of a single node, which offers both local access to secondary storage and processing power for image rendering. Our expectation is that good understanding of the client communication interactions with single-node servers will provide important experience towards building more complex servers consisting of multiple tiers.

5 Performance Evaluation

In our performance evaluation study, we consider the system throughput and turnaround time when creating maps with Mapserver by (a) running the application on the client and accessing the data from a local filesystem (Local), (b) running the application on the client and getting the data from an NFSv3-mounted remote file system (NFS), (c) invoking the application to run through *Lerna* on a remote file server that provides local access to the datasets (Lerna) (Figure 3). We report how the system resource utilization is affected by system parameters related to the load of the workload, and the features of the communication system.

5.1 Experimentation Environment

All the nodes that we used are Dell PowerEdge 4400 servers with 733 MHz Intel Xeon processors and 256 MB main memory, running FreeBSD 4.5. Each node is equipped with 100 MBit/s and 1 GBit/s network cards, and is connected to multiple Seagate Cheetah 10 KRPM 18.2 GB SCSI disks over two 160 MB/s channels. Each disk has nominal formatted internal transfer rate within the range 26-40 MByte/s, but practically the individual disk throughput drops to about 10 MB/s when head seeks are involved. We control the round-trip delay of the client/server path through Dummynet [23] running on the server. The *Lerna* implementation is based on SFS version 0.7.2. In our experiments, we also use Network File System version 3. We set the receiving socket buffer space of the client and the sending socket buffer space of the server to 512 KB.

We used nine geographical datasets made available by the U.S. National Survey (Table 1). One dataset is a topographic map of North America in colormapped GeoTIFF raster format occupying 71 MB. The remaining eight datasets contain U.S. geographical information in ESRI Shapefile vector format and have size between 7 MB and 71 MB. Projection to a common system of coordinates allows aligned data merging of different layers into a single map.

The queried images are requested in compressed PNG (Portable Network Graphics) format, have 600x300 pixels and occupy map space 20x10 degrees. Each query involves generating an image map according to a text configuration and scanning the returned image on the client side. The queries are uniformly distributed over the entire region of the North America raster, and have two layers. One layer is always the shaded relief, and the other is randomly chosen among the remaining eight datasets. The datasets are distributed across four different disks. Only part of the total

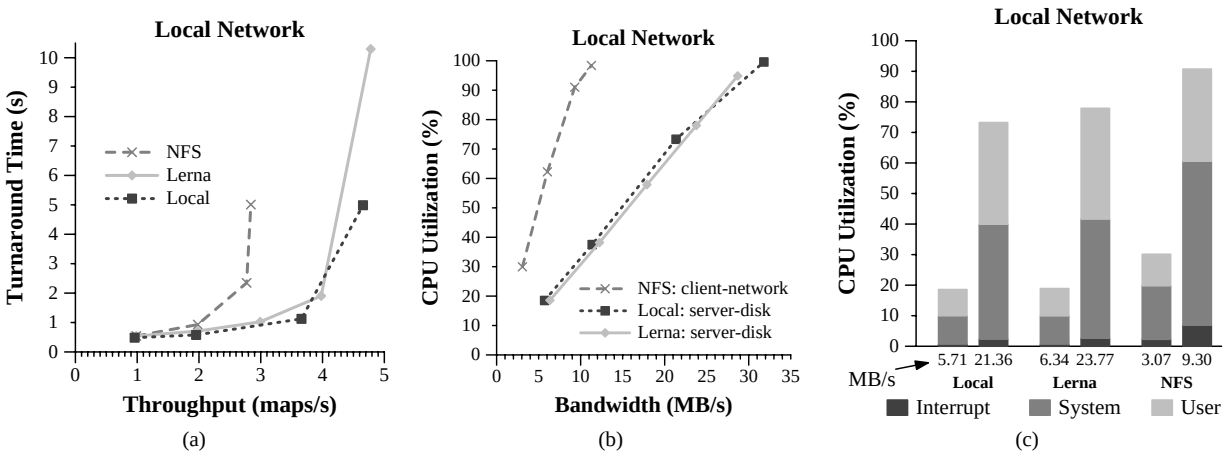

Figure 4. (a) We show the average turnaround time of a map request across different measured throughput values. We observe that when data are accessed remotely through NFS, the turnaround time starts to increase significantly at lower throughput than the Local and *Lerna* cases. (b) On the x-axis we illustrate the rate at which the computation node consumes data. We measure the network bandwidth in the NFS case, and the disk bandwidth in the Local and *Lerna* cases. On the y-axis we show the processor utilization of the computation node. Note that the computation node is the client in the NFS case, and the storage server in the Local and *Lerna* case. (c) We break down the processor utilization of the computation node into interrupt, system, and user activity at two different bandwidth values. *Lerna* and Local are comparable across the different types of activities. Instead, NFS incurs higher system and interrupt overhead at significantly lower data rates.

storage footprint of the datasets fits within the real memory of the server thus creating an I/O intensive environment. Map requests arrive to the system following a Poisson process. We measure the average amount of time that it takes to complete a map generation request (*turnaround time*) and the number of requests completed during an experiment run (*throughput*). Each run lasts 10 minutes, and the measurements of the first three minutes are discarded. Independent runs of each experiment are repeated (up to 30 times) until the half-length of the 95% confidence interval of the turnaround time lies within 5% of its estimated mean value.

5.2 Local Network Measurements

We begin our performance experimentation by having the client and server nodes of NFS and *Lerna* communicating over gigabit switched Ethernet at the minimum round-trip delay of about 150 μs. The Local case has both the server and client software running on the same node. We investigate how the Local, NFS, and *Lerna* system configurations behave at different map request rates. Depending on the node at which the computation takes place, we correlate the processor utilization with the corresponding rate of data that the processor consumes. Therefore, for NFS we measure the client processor utilization and the network bandwidth. Note that the reported network bandwidth is higher than the utilized disk bandwidth due to the benefits of caching in the NFS server. Instead, for the *Lerna* and Local cases, we measure the server processor utilization and the total disk bandwidth utilized.

From Figure 4(a) we observe that the average turnaround time of NFS starts to increase at lower throughput than Local and *Lerna*. In addition, Local and *Lerna* behave similarly up to the point that the server processor starts to saturate. Then *Lerna* shows higher turnaround time than Local. In Figure 4(b) we compare the processor utilization of the computation node with the consumed data bandwidth across the different configurations. We notice that NFS requires significantly higher processor utilization for consuming lower data bandwidth. This also explains the higher turnaround time of NFS in comparison to the other two cases. In particular, NFS saturates the client processor at roughly 10 MB/s, while Local and Lerna saturate the processor when consuming more than 30 MB/s.

In Figure 4(c), we further break down the processor utilization into interrupt, system and user activities. Even though the system overhead is the dominant activity in all cases, we show that NFS takes 50% more processing utilization for less than half of consumed data bandwidth. In addition, NFS demonstrates more than double the interrupt processing activity at significantly lower data bandwidth. On the other hand, both Local and *Lerna* require the same processing utilization at all three types of activities.

We attribute the increased system overhead of NFS to the network protocol processing involved in accessing each data block scanned by the processor. Current file systems and hard disks have the capability to recognize sequential access pattern and initiate readahead that fetches data blocks into the buffer cache before being actually requested by the processor. For both NFS and UFS we used the default parameters for readahead. However, as we describe in Section 5.4, increasing the readahead or the blocksize of NFS doesn't seem to improve the performance sufficiently

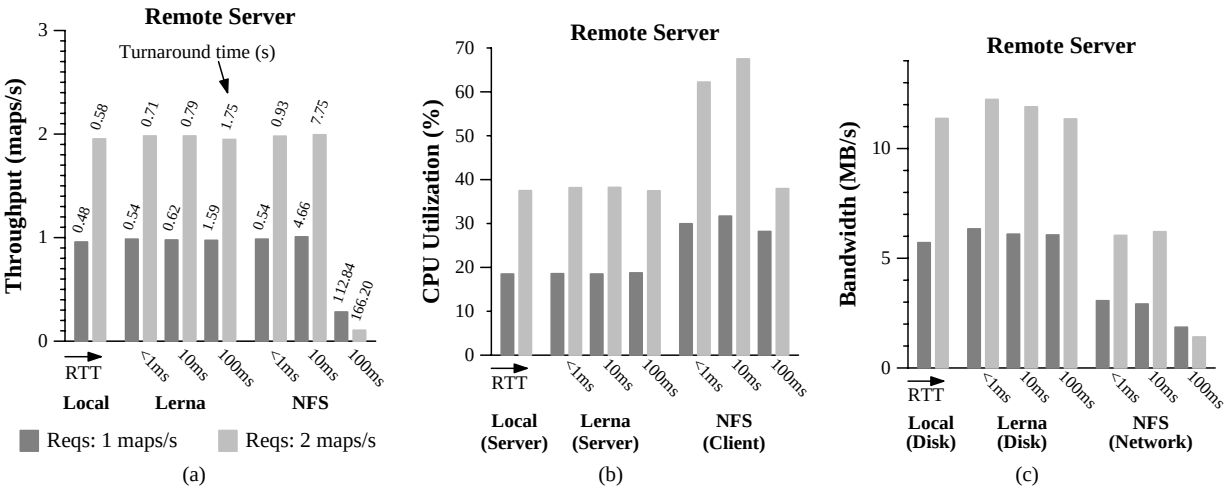

Figure 5. (a) We examine the impact of the client-server round-trip delay to throughput and turnaround time (shown at the top of the bars) at different map request rates. We observe that the turnaround time of *Lerna* remains less than 2 seconds even at 100 ms. Instead, the turnaround time of NFS-based computation increases linearly as a function of the network latency to several minutes. (b) As the round-trip delay increases, the processor utilization of the server remains the same in the *Lerna* configuration, and equal to the Local case. Instead, the processor utilization of the NFS client is higher at low network latencies but tends to drop as network latency increases. (c) It is notable that the utilized disk bandwidth remains the same between Local and *Lerna* at different round-trip delays, while the network bandwidth of NFS drops significantly.

to bring it into par with the other two access configurations.

Even though protocol transformations within the SFS toolkit are partly responsible for the *Lerna* processor utilization, we observed the SFS server daemon to occupy only 2-3% of the processor. The SFS overhead is involved only during the transfer of a map request to the server and the return of a complete map file to the client, rather than during every data block transfer that would be the case with plain NFS. Overall, we note that, at minimal round-trip delay in the client/server path, *Lerna* demonstrates a behavior that is similar from several aspects to the Local case, unlike NFS that is penalized by the overhead involved in individual data block transfers. The advantage of NFS from double caching space across the server and the client does not seem to create a measurable benefit in our experiments.

5.3 Effects of Server Proximity

Arguably, remote access of data over a wide-area network can adversely affect the efficient execution of applications. This is true both because fine-granularity accesses can be delayed by long latencies and network protocol processing, and because local caching of large data amounts occupies valuable resources. In the present section, we verify that Mapserver performs poorly as the network latency from the NFS server increases. In contrast, running applications close to the data is both feasible and beneficial. In Figure 5(a), we illustrate the turnaround time for generating two-layer maps when the processing takes place either on a remote server through *Lerna*, or locally with the dataset accessed from a remote server via NFS. For comparison purposes, we also include the Local case, where the client and server run on the same host.

From Figure 5(a) we notice a dramatic increase in the turnaround time of NFS-based processing as the network delay changes from < 1 ms to 100 ms. One main reason for this behavior is that Mapserver lacks explicit prefetching or asynchronous I/O for improved concurrency between communication and computation. Additionally, unpredictable data accesses would limit the potential benefit from compounding multiple operations into single requests supported by NFS version 4 [21]. Quite interestingly, *Lerna* performs similarly to the Local case when the delay is less than 1 ms, and at slightly higher turnaround time when the delay is 10 ms. Even when the network delay grows to 100ms, the turnaround time remains within a factor of three of the Local case. This increase is exclusively caused by transferring the requested maps from the server to the client, rather than the map processing that takes place within the server.

It is somewhat surprising, that at latency < 1 ms (Figure 5(c)), the network bandwidth consumed by the NFS client is about half the disk bandwidth consumed by Local and *Lerna*. Nevertheless, all three configurations achieve the same turnaround time. This indicates that the disk bandwidth utilized by Local and Lerna is unecessarily high. We believe that this is caused by the aggressive readahead automatically enabled within the disk and the local file system. We examine in the next section how manually increasing the readahead over NFS can affect the measured network bandwidth. In Figure 5(b), we also observe that the processor utilization of NFS is higher than that of Local but tends to drop as the network latency increases. Similarly, the network throughput of NFS is lower than that of Local, but

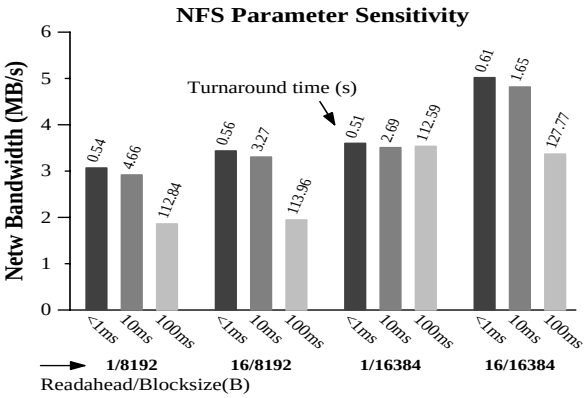

Figure 6. We examine the sensitivity of NFS performance to alternative transfer block sizes and readahead depths. We notice that even though aggressive prefetching helps improve network bandwidth, it is not sufficient to bring remote data access into par with remote application execution using *Lerna*.

drops drastically with longer delays (Figure 5(c)). Finally, the disk bandwidth utilized by *Lerna* remains insensitive to the delay increase, and the server processor utilization remains almost equal to that of the Local case across the different round-trip delays.

In order to explain the pathological behavior of NFS, we measured the system throughput and turnaround time along with the number of jobs concurrently running. We verified that the entire system can be treated as a queue that follows Little's law $L = \lambda R$, where L is the queue length, λ is the throughput and R is the turnaround time. Therefore, when round-trip delays increase the turnaround time, the multiprogramming level should also be high for the system throughput to remain the same. However, high multiprogramming levels make the system operation expensive due to the memory required to avoid thrashing. In fact, turnaround time grows dramatically as more jobs remain outstanding in a resource-constrained environment.

5.4 Effects of Blocksize and Readahead

In all our experiments with NFS, we used the UDP transport protocol with twenty `nfsiod` and `nfsd` daemons running on the client and the server, respectively. Having a large number of server daemons keeps the number of concurrent requests high, without necessarily improving the system performance. In the present section, we examine the performance sensitivity of NFS accesses to the transfer block size, and the readahead depth during block prefetching.

In Figure 6 we measure the network bandwidth and turnaround time with transfer block sizes of 8 KB (default) and 16 KB. Additionally, we examine the performance impact from setting the readahead depth to 1 (default) and 16. We consider all four combinations of the above parameter values, thus varying the total transfer size per read request between 8 KB and 256 KB. We notice that the larger the aggregate transfer size, the higher the measured network bandwidth. A noticeable exception to that is the case with round-trip delay equal 100 ms, where increasing the readahead depth leaves the network throughput unchanged. Furthermore, we observe a diminishing returns effect as the aggregate request size becomes larger. For example, increasing the request size by a factor of 32 only increases the measured bandwidth by a factor of two. This is reasonable given the fact that larger block sizes increase the contention for memory space in the client buffer cache.

The most important observation, however, is that the turnaround time (shown at the top of the bars) does not decrease considerably with increased prefetching, especially at long round-trip delays. We conclude that NFS tuning can have some positive impact to the NFS network throughput but this depends on the network latency and the amount of memory available for caching among other factors. Overall, we see inherent limitations in running applications that access data remotely, unless the applications themselves are modified for such operating conditions.

5.5 Microbenchmarks

In another set of experiments, we expose the amount of time spent at different parts of the system at low and high access loads. We control the demand for resources using alternative map image sizes requested from the server, while we keep the average request rate at 1 maps/second. The creation of the map description file involves LOOKUP, CREATE and WRITE RPCs, while the map access request involves LOOKUP and READ RPCs. These are only few of the remote procedure calls handled by the server.

In Table 2, we give a breakdown of the time spent on the server and the client during the prevalent remote procedure calls participating in accessing the mapserver through the file system interface. Map images of minimal size covering zero geographical area are requested from the server. It is notable, that the total time spent at user-level at both the server and the client is comparable to the delay of in-kernel server handling. We also measured the time spent within the network according to the payload size of the packets transmitted and received (shown between the parentheses). We report separately the amount of time required for invoking the mapserver application code on the server as a result of failed LOOKUP requests. However, we omit the latency on the client for transferring control from the application to the user-level file system daemon through the kernel. Since some system calls involve multiple remote procedure calls, inclusion of the above would require nontrivial instrumentation of kernel-level code, which was beyond our objectives

Call Type	User /ms (cln+srv)	Wire /ms	(Xmt/Rcv) /bytes	Kernel /ms (srv)
CREATE	0.465	0.197	(124/280)	0.760
LOOKUP	0.624	0.181	(144/184)	0.599
READ	0.698	0.312	(92/980)	0.602
WRITE	0.680	0.460	(1996/164)	0.668
app	142.246	0	(0/0)	0

Table 2. Breakdown of the time spent at different parts of the system across a subset of remote procedure calls and the application. A minimal image size spanning zero area across all the data layers is requested from the server. The user-level component refers to time on both the server and the client, while the reported kernel-level time corresponds to NFS handling on the server. In addition, we measure the time spent over the network for the payload size transmitted and received (shown in the parentheses).

Call Type	User /ms (cln+srv)	Wire /ms	(Xmt/Rcv) /bytes	Kernel /ms (srv)
CREATE	2.495	0.197	(124/280)	7.111
LOOKUP	3.828	0.181	(144/184)	9.776
READ	8.790	0.722	(92/5145)	7.842
WRITE	1.964	0.460	(1996/164)	7.634
app	1,562.350	0	(0/0)	0

Table 3. We examine the time components of specific RPCs and the application processing across our distributed testbed assuming image and mapped area parameters equal to those used in the rest of the study. In comparison to Table 2, we observe increased returned packet payload in the READ call (5145 bytes instead of 980), and a respective higher delay on the server and the client across all the calls. Nevertheless, the RPC overhead remains insignificant relatively to the application computation time.

in the current study.

In Table 3, we repeated the above experiments with map images of size and spanned area equal to that used in the rest of the study (specified in Section 5.1). The increase in the resource consumption on the server leads to an order of magnitude increase in the time spent on the server and the client for each reported remote procedure call. The average application invocation time itself increases by an order of magnitude from 142 ms to 1,562 s. However, the time spent on the network remains about the same except for the READ requests whose average reply payload size increases from 980 to 5145 bytes. Overall, we found the file system interface to incur an insignificant overhead when used by the client for triggering remote invocation of application computation on the server.

5.6 Accounting and Data Reorganization

In this section, we extend *Lerna* to accumulate usage statistics for the datasets used by the application services. We intercept each block request issued by Mapserver, and measure its average completion time in the local file system. We maintain the total number and average completion time of the block requests in a hash table indexed by filesystem identifier and file handle. Based solely on the statistics that we accumulate, we make decisions of how to best redistribute the datasets across the four disks in order to achieve load balancing. We validate our hypotheses by measuring the average utilized disk bandwidth on each disk. Note that the server processor utilization increases as a result of having each data block accessed through *Lerna* rather than directly from the local file system. Therefore, we expect to activate usage accounting only for tuning purposes rather than by default.

In Figure 7(a) we show the number of block requests across the nine datafiles. These measurements were recorded during an operation period of ten minutes at map generation rate of 0.5 requests/s. Each I/O request corresponds to an 8 KB data block. Note that the shaded relief is available on all four disks. In each map request we retrieve the shaded relief from the same disk that stores the other requested layer. Obviously, disks 3 and 4 received more block READ requests than disks 1 and 2 during our experiments. Consistently, we notice on Figure 7(b) that the utilized disk bandwidth of disks 3 and 4 is almost twice as high that of disks 1 and 2.

Based on the recorded block requests that each file received from Figure 7(a), we manually reorganize the datasets across the disks as shown on Figure 7(c). We notice that the total number of requests is now balanced across the four disks. Additionally, the measured disk bandwidth is about the same across the disks and roughly equal to 0.7 MB/s, as shown in Figure 7(d). We conclude that the file usage statistics made available by *Lerna* provide important information about the access frequency of the disks, which can be used for valuable data reorganization decisions. Arguably, the load-balancing problem is trivially solved in several cases by having the datasets stored on disk arrays or other types of storage systems, instead of manually distributing them across individual disks. On the other hand, it is very typical in large data centers to use multiple storage systems which make data distribution decisions unavoidable [2]. We believe that statistics gathering facilities of the type that we describe here should be standard functionality of commodity file system software in order to simplify and ultimately automate the solution of the data organization problem.

6 Related Work

In previous work, Wickremesinghe et al. introduced the notion of computation-enabled storage nodes called Active Storage Units (ASU) [27]. The present paper expands that work towards examining related system interfacing and per-

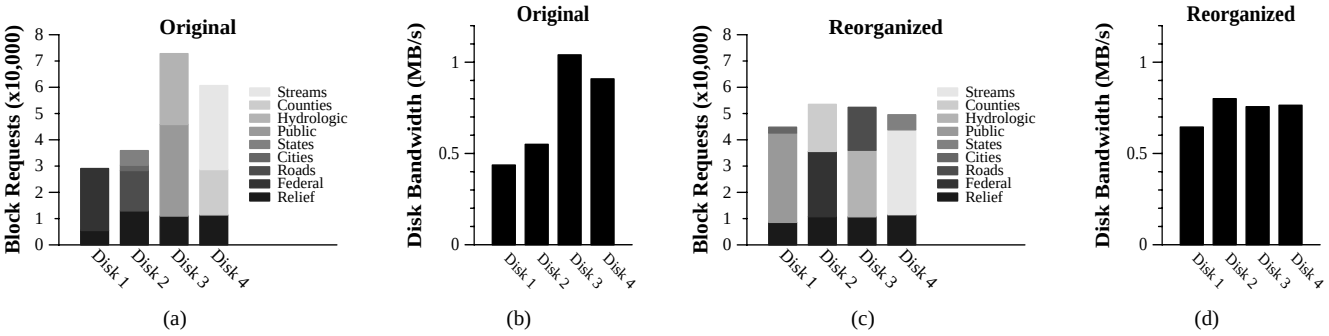

Figure 7. (a). Using the *Lerna* framework, we accumulate measurements of the block requests received for each dataset across the four disks. Note that the shaded relief file is accessed from the same disk that the other layer is requested on each map request. (b) We measure independently of *Lerna*, the utilized disk bandwidth across the four disks. The measured bandwidth is consistent with the block request measurement of (a). (c) By reorganizing the datasets across the four disks according to the block requests of Figure 7(a), we manage to keep the four disks almost equally utilized. (d). The measured bandwidth across the four disks after the reorganization validates the usefulness of the data gathered by *Lerna*.

formance evaluation issues in a file server accessed over a wide-area network.

Several research studies propose active disk systems that run application-level code on disk drives in order to reduce data traffic and improve parallelism [1, 22]. The performance and services of file server operations can be enhanced through a remote procedure call framework that allows running scripts securely on the server [24]. In clusters of clients and servers, application and file system functions can be partitioned and automatically placed on the server or the client according to dynamic application behavior and resource availability [3]. Services can be associated with virtual disks attached to block address ranges beyond the physical storage space of existing devices [16]. General data manipulation is possible through extra layers that can be statically embedded within the file system stack [4]. In the Hurricane File System complex file structures can be composed using simple building blocks [14]. Filter drivers can be used to trace system activity, but require tedious debugging in kernel mode [15].

The Data Grid architecture specifies several design principles for managing large data sets potentially distributed in a wide area [11]. Nest is a grid-enabled storage system that supports multiple data transfer protocols, models of concurrency, and request scheduling [6]. The Chimera architecture proposes a language to express derivations of relational data [12]. Datacutter is a middleware system that enables handling of spatial range queries and data filtering operations close to a multidimensional dataset archive [9]. The GASS service uses local reference counts to automate replication of a remote file into the local storage space [8]. LegionFS is an object-oriented file system that offers location-independent object naming, secure object access, system load distribution across multiple resources, and service extensibility [26]. The Storage Resource Broker middleware provides unified attribute-based access to data and metadata that are distributed across file systems, databases, and archives [5].

Middleware systems have been proposed by the database research community to allow integrated data access across multiple types of data repositories through a unified object-oriented data model [20]. The SDSS Science Archive provides access to astronomical object information via a three-tier architecture consisting of a user interface, an intelligent query engine, and a data warehouse [25]. Instead, we focus on specifying the systems infrastructure to facilitate remote access of data without enforcing a particular data model.

7 Conclusions and Future Work

In the present paper, we introduce a flexible system architecture that provides support for running application-specific computation within network file servers. We preserve the traditionally simple interface of file system accesses in order to invoke plug-in applications as a side-effect of regular I/O transfers. Additionally, we accumulate access performance measurements within the file server, and use them to effectively reorganize the stored data across multiple disks. We use a multi-layer mapping application to experimentally demonstrate significant performance advantages in comparison to alternative remote data access methods. We show that remote access of large datasets through traditional distributed file systems can significantly penalize the application processing performance, unless the unmodified application is enabled close to the stored data.

Our experience with the system so far has been very positive in terms of operation robustness and stability. In our future work, we plan to investigate the support of data streaming that will enable remote clients to gradually receive output as processing proceeds. Another issue is the decomposition of processing across multiple storage nodes, each adopting the *Lerna* internal organization. Finally, it is

important to examine the security implications of limiting the user-initiated computation through file access permissions.

8 Acknowledgments

The authors are thankful to Jeffrey S. Vitter for useful discussions on I/O requirements of external memory applications. They would also like to thank Christoph Spoerri for his feedback on current technology of geographical information systems.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ACM ASPLOS Conf*, pages 81–91, San Jose, CA, Oct. 1998.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conf*, pages 307–322, San Diego, CA, June 2000.

[4] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *USENIX Conf of File and Storage Technologies*, pages 129–145, San Francisco, CA, Mar. 2004.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *IBM CASCON*, Toronto, ON, Nov. 1998.

[6] J. Bent, V. Venkataramani, N. Leroy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *IEEE Intl Symp on High Performance Distributed Computing*, pages 3–12, Edinburgh, Scotland, July 2002.

[7] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url). In *Request for Comments: 1738*. Network Working Group, Dec. 1994.

[8] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. In *ACM Workshop on I/O in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, May 1999.

[9] M. Beynon, R. Ferreira, T. Kure, A. Sussman, and J. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symp on Mass Storage Systems*, pages 119–133, College Park, MD, Mar. 2000.

[10] I.-M. A. Chen, A. Kosky, V. M. Markowitz, and E. Szeto. Constructing and maintaining scientific database views in the framework of the object-protocol model. In *Intl Conf on Scientific and Statistical Database Management*, pages 237–248, Olympia, WA, Aug. 1997.

[11] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[12] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Intl Conf on Scientific and Statistical Database Management*, pages 37–46, Edinburgh, Scotland, July 2002.

[13] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole. Semantic file systems. In *ACM Symp on Operating Systems Principles*, pages 16–25, Asilomar,CA, Oct. 1991.

[14] O. Krieger and M. Stumm. Hfs: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, Aug. 1997.

[15] J. R. Lorch and A. J. Smith. The vtrace tool: Building a system tracer for windows nt and windows 2000. *MSDN Magazine*, 15(10):86–102, Oct. 2000.

[16] X. Ma and A. L. N. Reddy. Implementation and evaluation of an active storage system prototype. In *Workshop on Novel Uses of System Area Networks (HPCA-8)*, Cambridge, MA, Feb. 2002.

[17] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *USENIX Annual Technical Conf*, pages 1–14, Monterey, CA, June 2002.

[18] Mapserver 3.6. Environmental Resources Spatial Analysis Center, University of Minnesota, http://mapserver.gis.umn.edu/.

[19] D. Mazieres. A toolkit for user-level file systems. In *USENIX Annual Technical Conf*, pages 261–274, June 2001.

[20] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE Intl Conf on Data Engineering*, pages 251–260, Mar. 1995.

[21] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *Intl System Administration and Networking Conf*, page 97, May 2000.

[22] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[23] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1):31–41, Jan. 1997.

[24] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving rpc for active storage. In *ACM ASPLOS Conf*, pages 264–276, San Jose, CA, Oct. 2002.

[25] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *ACM SIGMOD Conf*, pages 451–462, Dallas, TX, May 2000.

[26] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. Legionfs: A secure and scalable file system supporting cross-domain high-performance applications. In *ACM/IEEE Conf on Supercomputing*, pages 59–59, Denver, CO, Nov. 2001.

[27] R. Wickremesinghe, J. S. Chase, and J. S. Vitter. Distributed computing with load-managed active storage. In *IEEE Intl Symp on High Performance Distributed Computing*, pages 13–23, Edinburgh, Scotland, July 2002.