The Cost of Transparency: Grid-Based File Access on the Avaki Data Grid

H. Howie Huang, Andrew S. Grimshaw

Department of Computer Science, University of Virginia
151 Engineer's Way, Charlottesville, Virginia 22904
{huang, grimshaw}@cs.virginia.edu

Abstract. Grid computing has been a hot topic for a decade. Several systems have been developed. Despite almost a decade of research and tens of millions of dollars spent, uptake of grid technology has been slow. Most deployed grids are based on a toolkit approach that requires significant software modification or development. An operating system technique used for over 30 years has been to reduce application complexity by providing transparency, e.g. file systems mask details of devices, virtual machines mask finite memory, etc. It has been argued that providing transparency in a grid environment is too costly in terms of performance. This paper examines that question in the context of data grids by measuring the performance of a commercially available data grid product – the Avaki Data Grid (ADG). We present the architecture of the ADG, describe our experimental setup, and provide performance results, comparing the ADG to a native NFS V3 implementation for both local and wide area access cases. The results were mixed, though encouraging. For single client local file operations, native NFS outperformed the ADG by 15% to 45% for smaller files, though for files larger than 32 MB ADG outperformed native NFS.

1 Introduction

Grid computing has become a popular and much-hyped term in both academia and industry, has been the subject of numerous research projects, and has been embraced as a key technology by major software vendors such as IBM, HP, Oracle, Sun, and others. Today, grid systems are still an active area of computer research while at the same time software based on grid technology has spread into production use in industries as diverse as pharmaceuticals, aerospace, financial services, and telecommunications, for sharing enterprise resources [1-3].

Grid computing is a much misunderstood concept. Many believe that it is primarily or even solely concerned with high performance execution environments – i.e. ones dedicated to improve the performance of high performance/parallel applications such as large individual MPI applications, or high throughput applications such as SETI@home and parameter space studies. However, the full scope and potential of grid computing is much greater than simply coupling together CPUs to provide more collective cycles. Rather, it is a concept that focuses system design on providing secure sharing and easy, virtualized access to resources of many types, e.g. CPU, storage, data, applications, policies, instruments, etc., across wide-area networks and across organizational boundaries [4-7].

While the majority of the early research and development of grid systems focused primarily on developing advanced execution environments [8-11], a growing number in the grid community have realized that it is desirable – indeed it is potentially even more useful – to develop equally sophisticated mechanisms for securely sharing and accessing data resources within and across organizations and to integrate such data resources into grid systems. There are several projects [12-20] - past and present, academic and industry - that focus on the data side of grid systems, developing what we call data grids.

Data grids take a number of different forms, and address data management issues such as transport, storage, wire integrity, security, replication, caching, and consistency. There are copy-in/copy-out systems that use FTP-like [21] mechanisms to make copies of the data at different locations, library-based approaches that provide special APIs to access remote data [13, 22, 23], complete distributed file systems [24], and interposition agents that mimic standard operating system behavior and redirect selected I/O operations to remote sources [9].

Each of the styles imposes different constraints, performance trade-offs, and programming burden on users. The most commonly used style to date has been copy-in/copy-out approaches as exemplified by the use of GridFTP [19]. The basic idea is simple. Before program execution, required data sets (including possibly the executable) are first copied from a remote location to a local file system. The application is executed, and specified output files are either copied to a remote location or are kept for local consumption.

There are several shortcomings to FTP-like approaches. Among them are redundant file copying, the need for accounts at multiple locations (if non-anonymous access is to be provided), and the cognitive burden placed on programmers to explicitly manage data transport, caching, and consistency issues. (A more detailed analysis is provided in Section 5.)

An alternative to FTP-like approaches is a Transparent Grid Data Management System (TGDMS) that emulates standard file system behavior while managing security, transport, caching, and consistency on behalf of the user. In a TGDMS the data resources in the grid are mapped into a directory-based namespace, which in turn is mapped into local operating system file system name spaces as a "mounted" directory. The result is that user applications can access data located throughout the grid without modifying their applications or scripts at all. Both programmers and end-users are completely isolated from even knowing the grid is there.

A common critique of TGDMSs is that the performance penalty for transparency is too high – and that it is not worth the cost. While "too high" is a subjective issue, the costs and performance penalty can be described quantitatively. In this paper we examine the performance of a commercially-available TGDMS, the Avaki Data Grid version 5.1[1], using a set of synthetic benchmarks that were designed to mimic four typical data access scenarios: a single client reading a local file, a single client reading a remote file, a single client writing a local file, and multiple parallel clients accessing a local file. The results were compared to the use of a Linux native NFS implementation. The outcome was that transparent read access can be achieved with either a small penalty (<10%) or – in the case of parallel access – a performance benefit. The story with write performance was not as promising - the penalty for writes was quite significant, typically a factor of eight slower. However, we believe that much of that can be attributed to implementation issues, and that more acceptable performance is possible.

The remainder of this paper is organized as follows: in Section 2 we present a brief description of the Avaki Data Grid architecture while the experimental design is discussed in Section 3. Performance results are detailed in Section 4. Section 5 presents related work and alternatives to TGDMSs.

2 The Avaki Data Grid

The Avaki Data Grid (ADG) [11-14][5, 14, 25, 26] is commercial grid software designed to simplify provisioning, access, and integration of data from multiple, heterogeneous, distributed sources. The primary goals of the Avaki Data Grid are 1) to deliver data to applications and users from a collection of geographically-distributed sources, and 2) to provide a single transparent interface to the data, facilitating data access, integration, and application development. Conceptually, the ADG architecture consists of three layers - an access layer, a transformation layer, and a provisioning layer.

The access layer provides a means for users and applications to access data in a location transparent fashion. This layer presents a familiar hierarchical namespace, where the interior nodes are directories, and the leaves are data objects – specifically files in our discussion here. Read/write access to files is provided via Avaki-specific NFS and CIFS servers, as well as via web interfaces, i.e., a browser. The NFS and CIFS servers are implemented by Avaki Data Grid Access Servers or DGAS. A DGAS presents the illusion of a single file system to client machines. By mounting the Avaki NFS or CIFS server, a client operating system essentially maps the Avaki namespace into the local operating system namespace, providing transparent access to remote resources. The DGASs are responsible for enforcing access control, caching data, and providing cache consistency.

The transformation layer supports the creation of abstract data objects, files, DBs, etc., that are the result of performing a transformation on other data objects. For example, filtering a file, merging and sorting two files, or performing an XSLT transformation. These transformations can be chained, and all of the

[1] Newer versions are now available – we believe that the main conclusions of this paper remain true.

resulting and intermediate data objects may exist in the namespace, have access control, and other metadata.

The provisioning layer maps local data sources, for example, file system directory trees, database tables and views, into the Avaki namespace. After checking permissions, the provisioning layer services data requests from the access and transformation layers.

Avaki has a grid-of-grids architecture. Each organizational unit, department, school, division, or company has one or more Avaki Grid domains. Grid domains are typically synonymous with administrative domains, i.e., authentication and identity domains. Avaki grid domains may be joined together by connecting their namespaces and exchanging certificates. In the case that two domains are separated by a firewall, a proxy server can be set up, which routes the requests and responses between the provider and the consumer.

An ADG domain consists of four components: a grid domain controller (GDC), which manages the global namespace; one or more grid servers (GS), which handle transforms, meta-data, and the data catalog; share servers (SS), which export local files and directories into the ADG in the form of Avaki shares; and data grid access servers (DGAS), which provide access to the data in the grid, as well as data and meta-data caching. An example of an ADG domain is given in Fig. 1.

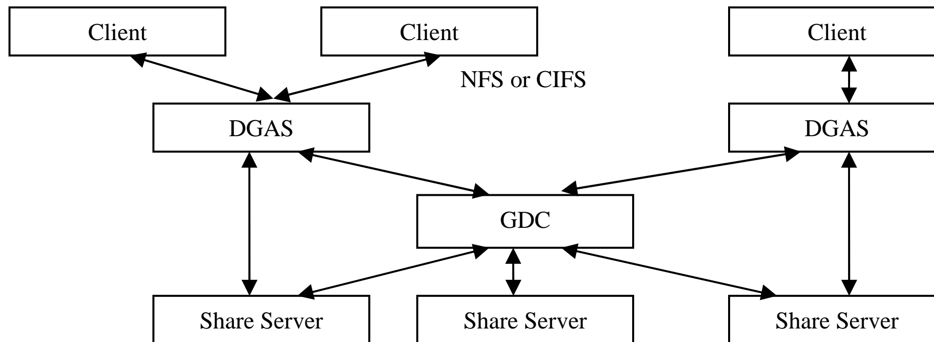

Fig. 1. Avaki Data Grid Architecture. A typical ADG domain consists of share servers that provision the data, a grid domain controller and zero or more grid servers that implement the name space and perform transformations, and one or more data grid access servers that provide cached access to client machines via NFS and CIFS

Tracing a single read through the stack is illustrative of how the pieces work together. Suppose a client application issues a read on a file. The read is routed to the local operating system NFS or CIFS client. Assuming that the data are not in the local cache, the request is forwarded to a DGAS which first checks permissions. Access control information is a part of the file metadata. If the file metadata are not in the local DGAS cache, then the DGAS interrogates the GDC using the users' credentials, asking for the metadata and for a signed access cookie[2]. The GDC looks up the location of the data object, loads metadata from a database, and checks the credentials against the access control list. Once the DGAS has the metadata and signed cookie, it interacts directly with the share server for all subsequent operations. The DGAS then reads the data from the share server and returns the results to the client operating system, which in turn returns the data to the application. Note that the above steps are not necessary if the data are cached.

In order to expedite the data access, the DGAS has both an in-memory cache and disk cache. On reads the DGAS checks to see if the data are already in one of its caches and still within the *coherence window*. If so, the cached data are returned. The coherence window is a property of each data object and is specified in seconds. If the data in the cache have been there longer than the coherence window, the DGAS must check with the share server to see if the data have been modified (a last write timestamp is part of the file metadata). If the data have been modified, the current cached data are evicted, and new data are retrieved and placed in the cache. If the data have not been modified, the window is reset and the read satisfied from the cache.

On writes, the DGAS cache use write-through; that is, the data are immediately written to the destination. The DGAS caches files in blocks. The block size, along with other DGAS parameters, can be

[2] The details of mapping from local identity spaces, e.g., Unix UID, to global name spaces, acquiring credentials, etc., is beyond the scope of this paper. Here we are giving a simplified overview that focuses on the data movement, not on security.

dynamically tuned at run-time. The DGAS evicts cache blocks based on an LRU policy when the cache is full.

3 Experiment Design

3.1 Objectives

Our experiment was designed with three objectives. First, the experiment should reveal the file I/O performance of the Avaki Data Grid, namely, bandwidth and latency. Bandwidth was defined as the bytes read or written by one or more clients over a time period. Latency was the time from when a request was submitted until the response was available.

Second, only the performance of the ADG should be measured; that is, the experiment should exclude caching effects from the local file system. Towards this end, we intentionally flushed the cache by disconnecting the NFS mount point after the previous I/O operation and reconnecting the mount point before the next I/O operation. Without the calls of *umount* and *mount* to flush caches, the clients would see the results nearly instantaneously from the local file system caches.

Third, we wanted to measure the performance of the ADG under stress, i.e., how well the ADG responded when many clients sent requests at the same time. We did this using an MPI application described later. In this case, the aggregate bandwidth was defined as the bytes received by all clients before the slowest client finished its request. Thus, it was possible that some clients finished their tasks early and waited for the completion of the slowest client. We computed the aggregate bandwidth as the product of the number of clients and the bandwidth of the slowest reader, since each reader read the same file. This penalized ADG, but for a parallel application, it is more realistic since most parallel applications must wait for the slowest task.

The experiments were characterized from the perspective of the client. During the read experiments, the client read data from the DGAS via the local OS and the NFS protocol and simply discarded the transferred data. On writes, the client transferred data from its memory to the DGAS via the local OS and the NFS protocol. Both read and write operations were sequential whole file operations, i.e., the client accessed a file from beginning to end. Given the elapsed time and the file size, we could compute the latency and the bandwidth for the client.

The read/write operation was repeated immediately. The first operation was intended to measure the performance when the DGAS cache was cold while the second operation measured performance when the DGAS cache was warm. Note that the local OS caches were emptied between runs as mentioned above.

3.2 Experiment Implementation

There were several factors that could affect performance. We focused on the three that we observed had the largest impact - remote versus local data, file size, and the number of concurrent readers.

For our tests we generated files containing synthetic data. The files varied in size from 1 MB to 1 GB by powers of 2, i.e., 1, 2, 4, 8 MBs. We placed those files in share servers both locally and remotely.

To measure bandwidth we created a synthetic benchmark to read/write data from/to a file. Two versions of the benchmark were created – a sequential version and an MPI parallel version. Unbuffered I/O functions were used in order to eliminate the effects from buffered I/O utilities. Bandwidth was measured by dividing the bytes read or written by the elapsed time. Pseudo code for the body of the sequential benchmark is as follows:

```
long start_times[2], end_times[2];
get_timing(start_times);                         // start the timing
while (result = read(fd,buf,buffersize) > 0); // read the file
get_timing(end_times);                           // stop the timing
```

For any two consecutive reads/writes the code was repeated. Between two requests the local caches were flushed by unmounting and mounting the NFS mount point.

The second benchmark measured multiple readers within a cluster. We slightly modified the sequential benchmark to add an MPI barrier before and after each read operation, which released all clients at the same time. To compute bandwidth we multiplied the number of MPI tasks (readers) by the file size, and divided by the elapsed time. Pseudo code for the parallel benchmark follows:

```
long start_times[2], end_times[2];
MPI_Barrier(MPI_COMM_WORLD);      // wait for everyone is ready
get_timing(start_times);                        // start the timing
while (result = read(fd,buf,buffersize) > 0); // read the file
MPI_Barrier(MPI_COMM_WORLD);      // wait for everyone completes
get_timing(end_times);                          // stop the timing
```

C was used for the implementation. The code was compiled by gcc with default settings.

4 Performance

The performance evaluation environment consisted of two ADG domains. Avaki Data Grid release 5.1 was used. One domain was *TACCDataGrid*, at the Texas Advanced Computing Center (TACC) of the University of Texas at Austin, whose grid server was hosted by a dual-processor 3.06 GHz Intel Xeon CPU with 2 GB memory, running version 2.4.20-31.9smp of the Linux kernel. The other domain was *UVaCenturionGrid*, at the University of Virginia, which had a dual-processor 2 GHz AMD Opteron CPU with 2 GB memory, called *centurion-home*, and a cluster of 20 machines called *sunfire1*, *sunfire2*, etc. Each *sunfire* was a dual-processor 2.8 GHz Intel Xeon with 1GB memory. All machines at the University of Virginia were running version 2.4.20-8smp of the Linux kernel. The two sites were connected via the Internet. Wide area throughput was limited by an OC-3 connection from the University of Virginia to the Internet and a 100 Mb connection from *UVaCenturionGrid* to the University of Virginia campus backbone.

We present and analyze the performance of the Avaki Data Grid under five scenarios. When we use the phrase “native NFS” in this paper, we mean the NFS V3 protocol with a native Linux NFS server. Avaki numbers are given using a Linux NFS client and an Avaki DGAS speaking NFS. For each scenario we ran the test 10 times before we computed the means and 95% confidence intervals.

Scenario 1: File open in LAN. The *centurion-home* hosted the GDC, a share server, and the DGAS. The reader, *sunfire1*, mounted the DGAS via NFS protocol just like any NFS client. In this scenario the DGAS handled requests as an NFS server. They were connected by Gigabit Ethernet. The latency for file open was measured.

Scenario 2: File read in LAN. The computers were connected as in Scenario 1. In this case, the bandwidth for a file read was recorded instead of latency. We read each file twice to reveal the effect of the cache in terms of bandwidth, and recorded both values.

Scenario 3: Simultaneous file read in LAN. The *centurion-home* hosted the grid server and share server. The difference was that more than one client read the file simultaneously. To be precise, 16 *sunfires* acted as concurrent clients. Each client had its own DGAS mounted locally via NFS protocol, i.e., each host had a separate DGAS that the host mounted and each DGAS communicated with the share server. All connections were Gigabit Ethernet.

Scenario 4: File write in LAN. Same as in Scenario 2, except that *sunfire1* became a writer. Because DGAS uses a write-through cache strategy, the write operations were performed only once.

Scenario 5: File read in WAN. Each domain had its own GDC. The source share server was hosted at TACC, as were the files to be read. The client, *sunfire1*, mounted to the DGAS on *centurion-home* via NFS. On read, *sunfire1* sent the read request to *centurion-home*, which forwarded the request to the grid server at TACC. In response, the share server from TACC sent the data back to *sunfire1* provided the user had permission to the data. We do not show in the paper the performance of file writes in WAN because the performance as expected suffered by the limited bandwidth and long latency of wide area network.

4.1 File open in LAN

Latency was measured as the time for a client to establish a connection and open a file on a server. The client opened an existing file for reads. In case of writes, the client created a new file. Both the client and the server were in the local area network. Table 1 lists the file open latencies for native NFS and ADG in LAN. A file open for read in ADG needed three times as much time as native NFS, while a file open for write in ADG imposed even more significant overhead compared to NFS. We speculate that the reason is that ADG has to perform many steps with regard to metadata at several locations, such as share server, data grid access server and grid server. In contrast, metadata in NFS is handled at one location, the NFS server. As a result, NFS consumed less time and was more efficient.

Table 1. File open latencies. We ran the tests for ten times

Read/Write	Latency (milliseconds)	95% Confidence Level
Native NFS read	1.1112	0.0385
Native NFS write	2.2947	0.1202
ADG read	3.5117	1.4938
ADG write	28.6897	11.9093

4.2 File read in LAN

Corresponding to Scenario 2, the result of file read by an NFS client across a LAN is shown in Fig. 2. We can see that the bandwidth of the second read achieved 60 MB/s, an increase as much as six times of 10 MB/s bandwidth of the first read. The performance of the subsequent read improved greatly because the files were cached in the DGAS after the first read. Such a bandwidth increase for the second read sustained till the file size exceeded 1 GB, which exceeded the cache size of the DGAS.

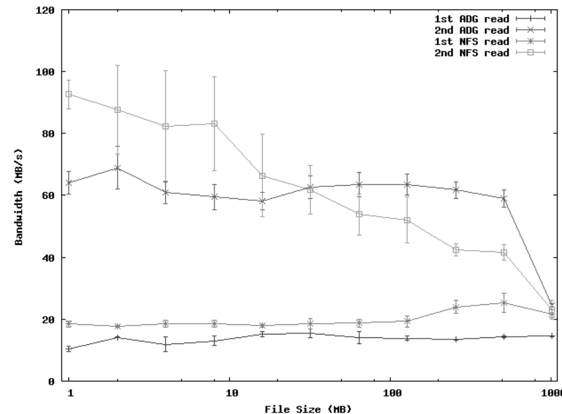

Fig. 2. File read in LAN for both ADG and native NFS. Average performance for each of the four tests is given, as well as the 95% confidence intervals

With the intention of finding out the performance penalty that ADG pays for transparency, we always measured the performance of native NFS in the same setting. Fig. 2 also shows the performance of a native NFS client reading a file from an NFS server. For the first read, native NFS beat ADG by a margin of 5 MB/s. However, ADG outperformed native NFS by up to 20 MB/s for the second read, sustaining the bandwidth for files that were larger than 8 MB. This indicated that native NFS has a smaller cache than ADG, which makes use of disk space as file cache in addition to in-memory cache. Although writing data to the disk slowed down the first read, such aggressive caching of DGAS significantly benefited the second read on large files, and for files that were being read from a remote location. The performance differences in percentages are shown in Table 2 below. One can see that the native NFS usually beat the ADG by at least 15%, except on the second read of larger files.

Table 2. Difference in Percentage (%) = (1 - ADG/NFS) * 100%

File Size (MB)	Difference in Percentage (%)	
	1st Read	2nd Read
1	43.72495	30.94852
2	19.83195	21.25632
4	35.95397	26.16955
8	29.6479	28.51128
16	15.19068	12.40469
32	16.30578	-1.28616
64	24.50852	-17.8668
128	28.36182	-22.0486
256	44.22514	-45.4627
512	43.55048	-42.122
1024	33.19589	-6.48357

4.3 Simultaneous file read in LAN

We chose a 128 MB file size for our simultaneous readers experiment. Both the performance of ADG and that of native NFS are shown in Fig. 3. It is well-known that native NFS does not scale when there are many concurrent reads. Our tests confirmed this fact: the aggregate bandwidth for NFS consistently stayed around 110 MB/s. So as more clients were added, each client got less bandwidth. When there were four clients, each one achieved a bandwidth of 27 MB/s but when there were 16 clients, each one only achieved about a bandwidth of 7 MB/s. In addition, there was no significant difference between the first and second read, which suggested that NFS cache has no effect for the second read because it is network bandwidth limited.

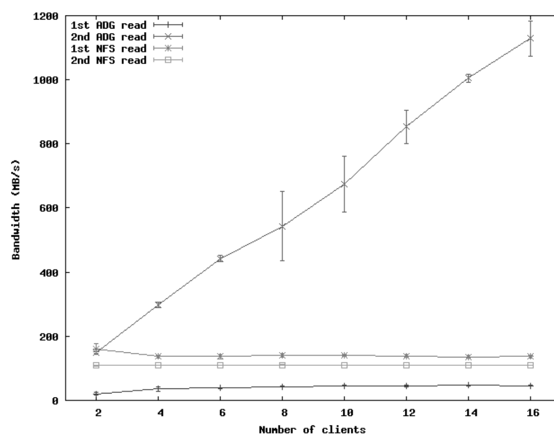

Fig. 3. Simultaneous file read. Averages are shown with 95% confidence intervals

Although the first read for ADG was slower than that of native NFS, each ADG client could expect a much better performance at the second read, thanks to ADG caches. In ADG, the aggregate bandwidth of the first read maintained at a level of 45 MB/s for up to 16 clients, where the bandwidth was 2.8 MB/s for each client. However, for the second read, the aggregate bandwidth was linear with the number of clients. The more clients were added, the larger became the aggregate bandwidth. When there were four clients, the aggregate bandwidth was 298 MB/s. When there were 16 clients, a four times increase in the number of clients, the aggregate bandwidth was 1128 MB/s, an increase of 3.9 times in bandwidth. On average each client got a bandwidth of 70 MB/s, eight times what a native NFS client got when there were 16 clients. This suggested that ADG scales well when there is reuse, as for example, when bioinformatics codes are being repeatedly run against a large sequence or protein database.

4.4 File write in LAN

For file write, both native NFS and ADG presented a fairly consistent performance for various file sizes shown in Fig. 4. In general, native NFS significantly outperformed ADG for write. A client could write a file at a bandwidth of 40 MB/s in native NFS, while at the bandwidth of 4 MB/s in ADG. ADG wrote data through to the share server without any caches. Thus, the performance suffered as more components were involved in ADG. We did not repeat file write twice as we did for file read because ADG directly writes data to the share server without any cache. Each time a client writes a file, he/she should expect the same bandwidth.

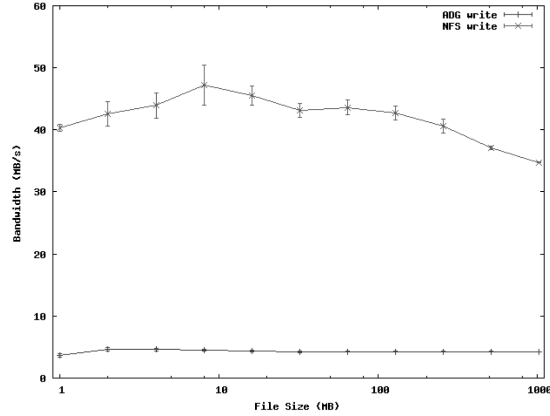

Fig. 4. File write in LAN. Averages are shown with 95% confidence intervals

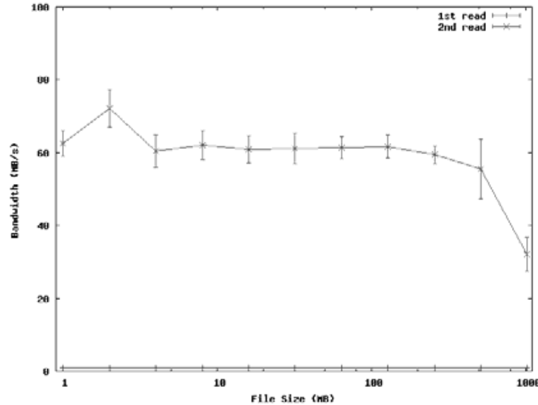

Fig. 5. File read in WAN. Averages are shown with 95% confidence intervals

4.5 File read in WAN

In Fig. 5 we present the bandwidth file read in the wide area network between the University of Virginia and TACC. As expected the first read was slow at a bandwidth close to 1 MB/s due to network limitations. The performance of the second read in WAN was around 65 MB/s, which was comparable to that in LAN because the data were cached in the DGAS after the first read. This characteristic of ADG can greatly reduce the read latency and network load in cases that a lot of clients want to read the same set of files on the remote location. After the data are cached in the DGAS, the Avaki Data Grid is able to serve many clients with the local copy of the data without the need for retrieving one copy for each client. This saves significant network bandwidth resources. Most importantly, a client is able to access the data quickly and seamlessly regardless of the original locations. We do not compare ADG with native NFS here, because NFS V3 is not suitable for wide area networks, which will be explained in Section 5.

5 Related Work

The problems that data grids solve have been around for as long as networks have existed between computers. A number of solutions have been created to solve the problems of remote data access. In this section, we take a closer look at some of the more popular solutions and present their advantages and disadvantages. For each solution we will take up a case of a local user at one site trying to share a file with a remote user at another site. The first user, say Alice, is a user on the local machines owned by one company, whereas the second user, say Bob, is a user on the remote machines owned by another company. The two companies may not share a mutually-trustful relationship although the sharing of Alice's file with Bob has been approved.

5.1 Network File System – NFS

NFS V3 is the standard Unix solution for accessing files on remote machines within a LAN. With NFS, a disk on a remote machine can be made part of the local machine's file system. Accessing data from the remote system now becomes a matter of accessing a particular part of the file system in the usual manner. In our use-case above, Alice could run an NFS server on her machine, and Bob could run an NFS client to mount Alice's file system onto his. Bob can now access the exact file that Alice wishes to share.

There are several advantages to NFS, the most significant of which is that it is easy to understand. Typically, Unix system administrators configure the server and client, and ordinary users like Alice and Bob simply use it without necessarily realizing that they are doing so. Moreover, applications need not be changed to access files on an NFS mount – the NFS server supports standard OS file system calls. Accordingly, files may be accessed entirely on in parts as desired. Finally, the NFS server and client tools come standard on all Unix's. On Windows, a special service pack must be purchased and installed.

The biggest disadvantage with NFS V3 is that it is a LAN protocol – it simply does not scale to WAN environments. If Alice and Bob are separated by more than a few buildings, using NFS between them becomes problematic. Moreover, if Alice and Bob belong to different organizations, as they are in our use-case, NFS cannot be deployed with reasonable guarantees of security.

Three characteristics of NFS doom it for use in wide-area, multi-organizational settings. First, the caching strategy on the NFS server typically releases data after 30 seconds and reloads the data on subsequent access. The result is a frequent retransmission of data and over-consumption of bandwidth. A related problem is that the read block size is too small, typically 8 KB. In a wide-area environment, latency can be high, therefore larger block sizes are needed to amortize the cost of the remote procedure call (RPC). Although the block size can be changed, most NFS V3 clients do not change it.

Second, and most seriously, NFS V3 does not address security well. An NFS V3 request packet is sent in the clear and contains the (integer) User ID (UID) and Group ID (GID) of the user making the read or write request. The NFS V3 server "trusts" the NFS client to not lie about the identity of the user making the request. Such a trustful relationship does not exist among multiple organizations, such as Alice's and Bob's. Even if the organizations do trust each other, man-in-the middle, imposter and snooping attacks can be made with NFS traffic. A VPN deployed between the organizations may attenuate some of these attacks, but VPNs introduce their own problems of management, trust and scalability. Further, firewalls typically do not permit NFS traffic through them making NFS impossible for cross-site use across firewalls.

Third, even assuming that the packets can be sent in a safe and trustworthy fashion, NFS requires that the identity spaces at the two sites be the same. In other words, not only should Alice and Bob have accounts on each other's machines, but Alice's UID on Bob's machine must be the same as her UID on her own machine. Likewise, Bob must synchronize his UIDs on his machine and Alice's machine in the same manner. Such synchronization would be possible if Alice and Bob were within a single domain; in our realistic use-case, they are not.

Other disadvantages plague NFS; we will mention them briefly here. NFS performance does not scale in a wide-area setting because it is a request-reply protocol which requires acknowledgments to be sent for every request, thus increasing effective transmission latency. NFS is a stateless protocol, i.e., the server does not keep track of the position of files being read. Accordingly, the server cannot pre-cache data or pre-position accesses to give clients better performance. Increasing the number of clients overwhelms the

one server deployed to serve data, thus reducing performance. In our use-case Alice wants to give Bob access to one of her files. If Bob also had some files he wished to share with Alice, he would have to run an NFS server on his machine and ask Alice to run an NFS client on hers. This kind of configuration can lead to a morass of cross-mounting, which can over-burden most administrators. In general, NFS requires $m \times n$ connections if m clients access data on n servers.

5.2 FTP and GridFTP

FTP has been the tool of choice for transferring files between computers since the 1970s. FTP is a command-line tool that provides its own command prompt and has its own set of commands. FTP may be used within a script; however, in that case, the password for the remote machine must be stored in a clear-text file on the local machine. Using ftp, Alice may connect to Bob, enter a username and password relevant to Bob's machine, change to the appropriate remote directory and then transfer the file.

The benefit of using ftp is that it is relatively easy to use, has been around for a long time and is therefore likely to be installed virtually everywhere. However, the disadvantages of ftp are numerous. First, Alice must have access to an account on Bob's machine, complete with username and password. Having such access means that Alice potentially could do more than just file transfer – she might be able to log in to Bob's machine and access files, directories and other machines to which she has not been given explicit access. From Alice's perspective, every transfer requires her typing the appropriate machine name, username and password. She could ameliorate some of this burden by using a configuration file for ftp, but that file may require storing a clear-text password for Bob's machine.

In order to eliminate some of these problems, Bob's site may choose to implement anonymous ftp. In this case, Alice need not have a username and password for Bob's machine, but must still remember the machine name and part of the directory structure. The problem with anonymous ftp is obvious – *anyone* may now access Bob's ftp directory, not just Alice. The potential for unauthorized overwriting or filling up of disk space is large.

FTP is also inherently insecure; passwords and data are transmitted in the clear. Snooping attacks may easily compromise Alice and Bob. Hence, most sites that have firewall protection shut down the standard ftp port to discourage such attacks. Even without firewalls, there are other disadvantages to using ftp. Because ftp requires making a copy of the data on Bob's machine, if Alice ever changes her own copy of the file, she must remember to ftp the new version of the file. Moreover, if Bob ever changes the file, he must remember to ftp the file back to Alice and reconcile concurrent changes, if any. This process is fraught with potential for inconsistencies, and the problem is compounded if additional people need to use Alice's file and receive versions from her at different points in time. Also, ftp is an all-or-nothing protocol – if even one bit of a large file changes, the entire file must be copied over. Finally, ftp is not conducive to programmatic access. Applications cannot use ftp to take advantage of remote files without significant modification.

SCP/SFTP belong to the *ssh* family of tools. SCP is basically a secure version of the Unix rcp command that can copy files to and from remote sites, whereas sftp is a secure version of ftp. Both are command-line tools. The syntax for scp resembles the standard Unix cp command with a provision for naming a remote machine and a user on it. Likewise, the syntax and usage for sftp resembles ftp.

The benefits of using scp/sftp are that their usage is similar to existing tools. Moreover, password and data transfer are encrypted, and therefore secure. However, a disadvantage is that these tools must be installed specifically on the machines on which they will be used. Installations for Windows are hard to come by. Moreover, scp/sftp do not solve several of the problems with ftp. In our use-case, Alice must still have access to an account on Bob's machine and she must continue to remember the appropriate machine name, username and password. Alice could ameliorate some of this burden by using an authorized keys file which permits password-less access, but she must then store her private key safely on her local machine.

Sites protected by firewalls may permit scp/sftp traffic on the designated port because the traffic is encrypted. However, scp/sftp does not attempt to solve the consistency problems of proliferating multiple copies of the file. Like ftp or rcp, a change of even one bit requires the entire file to be copied over. Finally, these tools are not conducive to programmatic access. Applications cannot take advantage of remote files using scp/sftp without significant modification.

GridFTP is a tool for transferring files built on top of the Globus toolkit [27]. GridFTP is an example of a service that characterizes the Globus "sum of services" approach for a grid architecture. Alice and Bob, in our use-case, could use GridFTP to transfer files from one machine to another, similar to the way they would use ftp. Naturally, both parties must install the Globus toolkit in order to use this service.

GridFTP solves the privacy and integrity of the problems with ftp by encrypting passwords and data. Moreover, GridFTP provides for high-performance, concurrent file transfer by design. An API enables accessing files programmatically, although applications must be re-written to use new calls. Data can be accessed in a variety of ways - for example, blocked and striped. Part or all of a data file may be accessed, thus removing the all-or-nothing disadvantage with ftp.

However, GridFTP does not address the identity space problems with ftp. Alice and Bob in our use-case must still have an account on each other's machine, thus giving them more privileges than just file access. Instead of a machine name, username and password as in ftp, Alice and Bob have to remember just the machine name. Their identities are managed by Globus using session-based credentials. Finally, GridFTP does not solve the problems of maintaining consistency between multiple copies, because Alice and Bob would still be required to maintain at least two copies of the file, one on each user's machine.

5.3 NFS over IPSec

IPsec is a protocol devised by IETF to encrypt data on a network. With IPSec installed and configured properly, all traffic on a network can be encrypted. Consequently, illegitimate snooping of network traffic does not affect the privacy and integrity of the communication between a server and a client. NFS over IPSec implies traffic between an NFS server and an NFS client over a network on which data has been encrypted using IPsec. The encryption is transparent to an end-user. NFS over IPSec removes some, but not all, of the disadvantages of using NFS.

NFS over IPSec results in encrypted NFS traffic, thus regaining privacy and integrity. However, NFS continues to be a LAN-based protocol which does not scale to the WAN-like environment typical in our use-case. All of the performance, scalability, configuration and identity space problems we discussed earlier remain. In addition, in order to deploy IPSec, all of the machines in Alice's and Bob's domains must be reconfigured. Specifically, their kernels must be recompiled in order to insert IPSec in the communication protocol stack. This recompilation is hard; anecdotal evidence suggests that the recompilation is risky, error-prone and ill-documented. Finally, once this recompilation is done, *all* traffic between all machines is encrypted. Even web, email and ftp traffic is encrypted whether desired or not.

5.4 De-Militarized Zone – DMZ

A DMZ is simply a third set of machines accessible to both Alice and Bob using ftp or scp/sftp, established to create an environment trusted by both parties. When Alice wishes to share a file with Bob, she must transfer the file to a machine in the DMZ, inform Bob about the transfer and request Bob to transfer the file from the DMZ machine to his own machine. Although both Alice and Bob have relatively unfettered access to the DMZ machines, neither party compromises his/her own machines by letting the other have access to them.

With a DMZ, neither Alice nor Bob requires an account on the other's machines. Typically, companies deploying DMZs also deploy scp/sftp or some such secure means of file transfer. Therefore, these tools must be installed on all concerned machines. Alice and Bob both have to remember machine names, usernames and passwords for the DMZ machines. However, they now have to remember an additional step of informing the other whenever a transfer occurs.

DMZs worsen the consistency problems by maintaining three copies of the file. Also, because the file essentially makes two hops to get to its final destination, network usage increases. DMZs may address security concerns, but they do not ameliorate any of the other problems with scp/sftp and they do increase administrative burden. If Alice's company decides to co-operate with a third company, thus requiring Alice to interact with Chris at that company, she must now create and remember yet another DMZ

configuration for interacting with Chris. The Alice-Bob DMZ cannot be reused because of the potential for Chris to access files intended for Bob.

5.5 Andrew File System – AFS

The Andrew File System is a distributed network file system that enables access to files and directories distributed across multiple sites. Access to files involves becoming part of a single virtual file system. AFS comprises several cells, with each cell representing an independently-administered file system. In our use-case, the file system on Alice's machine would be one cell, whereas the file system on Bob's machine would be another. The cells together form a single large virtual file system that can be accessed similar to a Unix file system.

AFS permits different cells to be managed by different organizations thus managing trust. In our use-case, Alice and Bob would not require accounts on the other's machines. Also, they could control each other's access to their cell using the fine-grained permissions provided by AFS. When Bob accesses one of Alice's files for which he has permission, he accesses exactly the current copy of the file. Thus, AFS avoids the consistency problems with other approaches using copy-on-open semantics unless there are multiple concurrent writers (which AFS does not deal with well.) In order to improve performance, AFS supports intelligent caching mechanisms. Since access to an AFS file system is almost identical to accessing a Unix file system, users have to learn few new commands, and legacy applications can run almost unchanged.

AFS implements strong security features. All data are encrypted in transit. Authentication is using Kerberos, and access control lists are supported.

The drawbacks of AFS revolve around the use of Kerberos and the fact that it is a file system. Let me explain. The use of Kerberos means that *all* sites and organizations that want to connect using AFS must themselves use Kerberos authentication *and* all of the Kerberos realms must trust each other. In practice this means changing the authentication mechanism in use at the organization. This is a non-trivial and typically politically very difficult step to accomplish. Second, the realms must trust each other. This is similarly difficult to accomplish. Third, the Kerberos security credentials time-out eventually. Therefore, long-running applications must be changed to renew credentials using Kerberos's API. Also, AFS requires that all parties migrate to the same file system. In other words, Alice and Bob would have to migrate their entire file systems to AFS, which would probably be a significant burden on them and their organizations.

6 Conclusion

The question we asked at the beginning of the paper is whether a grid file system can achieve transparency without significant compromise in performance. To answer it, we measured the performance of file I/O in the Avaki Data Grid and compared it to "native" NFS performance. The results were mixed, though encouraging. For single client local file operations, native NFS outperformed the ADG by 15% to 45% for smaller files, though for files larger than 32 MB ADG outperformed native NFS. For writes ADG was significantly slower than the native NFS – it was not quite clear why. On the other hand, for concurrent readers ADG outperformed native NFS by as much as a factor of five.

The big win is in remote data access. Avaki's cache makes subsequent access significantly faster than re-transmitting the data. While end-users could explicitly manage their own cache, in our experience they don't. The result is either significant resending of data that has not changed, or users accessing out of date data inadvertently[3].

In conclusion, although the data grid we tested did introduce some overheads, it does not seem, in our opinion, an unreasonable price to pay in order to achieve transparency and scalability. By quantifying I/O performance in this paper, we hope to help researchers and IT professionals gain an insight into the trade-offs between transparency and performance in data grids.

[3] The authors have seen an internal report of a top 10 pharmaceutical company that reports that 40% of all jobs end up being recomputed because the wrong – or more often old – data were used. Thus the jobs need to be re-executed, consuming resources, and delaying drug development.

Acknowledgments

We would like to thank several people for their help, without which those experiments could not have taken place: Scott Ruffner and Mark Morgan at the University of Virginia, Jay Boisseau at the Texas Advanced Computing Center of the University of Texas at Austin, Jerry Perez at Texas Tech University, and Chuck Kesler at North Carolina BioGrid.

References

1. Berman, F., G.C. Fox, and A.J.G. Hey, Grid Computing: Making the Global Infrastructure a Reality. Wiley Series in Communication Networking & Distributed Systems. 2003: Jon Wiley & Sons.
2. Abbas, A., Grid Computing: A Practical Guide to Technology and Applications. 2004: Charles River Media.
3. Foster, I. and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure. 1999: Morgan Kaufman Publishers.
4. Foster, I., et al., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. 2002.
5. Grimshaw, A.S., et al., From Legion to Avaki: The Persistence of Vision, in Grid Computing: Making the Global Infrastructure a Reality, Fran Berman, Geoffrey Fox, and T. Hey, Editors. 2003.
6. Grimshaw, A. and A. Natrajan, Legion: Lessons Learned Building a Grid Operating System. Transactions of the IEEE, 2005.
7. Lewis, M.J., et al., Support for Extensibility and Site Autonomy in the Legion Grid System Object Model. Journal of Parallel and Distributed Computing, 2003. Volume 63: p. pp. 525-38.
8. Frey, J., et al., Condor-G: A Computation Management Agent for Multi-Institutional Grids. Journal of Cluster Computing, 2002. 5: p. 237-246.
9. Thain, D., T. Tannenbaum, and M. Livny, Condor and the Grid, in Grid Computing: Making The Global Infrastructure a Reality, F. Berman, A.J.G. Hey, and G. Fox, Editors. 2003, John Wiley.
10. Foster, I. and C. Kesselman, Computational Grids, in The Grid: Blueprint for a New Computing Infrastructure. 1999, Morgan-Kaufman.
11. Grimshaw, A.S., The Legion Vision of a Worldwide Virtual Computer. Communications of the ACM, 1997. 40(1): p. 39-45.
12. Kola, G., et al., DISC: A System for Distributed Data Intensive Scientific Computing. in Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS'04). 2004. San Francisco, CA.
13. White, B., A. Grimshaw, and A. Nguyen-Tuong, Grid Based File Access: The Legion I/ O Model. in Proceedings of the Symposium on High Performance Distributed Computing (HPDC-9). 2000. Pittsburgh, PA.
14. Grimshaw, A., Avaki Data Grid - Secure Transparent Access to Data, in Grid Computing: A Practical Guide To Technology And Applications, A. Abbas, Editor. 2003, Charles River Media.
15. OGSA-DAI project home page. [cited; Available from: http://www.ogsadai.org.uk/.
16. White, B., et al., LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. in Proceedings SC 01. 2001. Denver, CO.
17. DAIS Working Group. [cited; Available from: https://forge.gridforum.org/projects/dais-wg.
18. OGSA Byte I/O Working Group. [cited; Available from: https://forge.gridforum.org/projects/byteio-wg.
19. Allcock, W., GridFTP Protocol Specification (Global Grid Forum Recommendation GFD.20). 2003 [cited; Available from: http://www.globus.org/alliance/publications/papers.php#GridftpSpec02.
20. Bester, J., et al., GASS: A Data Movement and Access Service for Wide Area Computing Systems. in Sixth Workshop on I/O in Parallel and Distributed Systems. 1999.
21. Postel, J. and J. Reynolds., RFC 959 - File Transfer Protocol. 1985.
22. Rajasekar, A., et al., Storage Resource Broker - Managing Distributed Data in a Grid. Computer Society of India Journal, Special Issue on SAN, 2003. 33(4): p. 42-54.
23. GGF, Simple API for Grid Applications (SAGA), Global Grid Forum.
24. Satyanarayanan, M., Scalable, Secure, and Highly Available Distributed File Access. Computer, May 1990. 23: p. 9-18,20-21.
25. Avaki, http://www.avaki.com/.
26. Avaki, Avaki Data Grid Administration Guide, Release 5.1. May 2004.
27. Globus, http://www.globus.org/.