Avaki Data Grid – Secure Transparent Access to Data

Andrew Grimshaw

Mike Herrick

Anand Natrajan

Background

"For over thirty years science fiction writers have spun yarns featuring worldwide networks of interconnected computers that behave as a single entity. Until recently such science fiction fantasies have been just that. Technological changes are now occurring which may expand computational power in the same way that the invention of desk top calculators and personal computers did. In the near future computationally demanding applications will no longer be executed primarily on supercomputers and single workstations using local data sources. Instead enterprise-wide systems, and someday nationwide systems, will be used that consist of workstations, vector supercomputers, and parallel supercomputers connected by local and wide area networks. Users will be presented the illusion of a single, very powerful computer, rather than a collection of disparate machines. The system will schedule application components on processors, manage data transfer, and provide communication and synchronization in such a manner as dramatically improve application performance. Further, boundaries between computers will be invisible, as will the location of data and the failure of processors." [1]

I think the quote is over-used. Besides, several of the phrases refer explicitly to compute-grids and have no meaning for data grids.

The future is now; after almost a decade of research and development by the Grid community we see Grids (then called Metasystems [3]) being deployed around the world both in academic settings, and more tellingly, in production commercial settings.

What is a Grid? What use is a Grid? What is required of a Grid? Before we answer these questions, let us step back and define what is a Grid and what are its essential attributes.

Our definition, and indeed a popular definition, is: A Grid *system* is a collection of distributed resources connected by a network. A Grid system, also called a *Grid*, gathers resources – desktop and hand-held hosts, devices with embedded processing resources such as digital cameras and phones, or tera-scale supercomputers – and makes them accessible to users and applications in order to reduce overhead and accelerate projects. A Grid *application* can be defined as an application that operates in a Grid environment or is "on" a Grid system. Grid system software (or middleware), is software that facilitates writing Grid applications and manages the underlying Grid infrastructure.

The resources in a Grid typically share at least some of the following characteristics:

- They are numerous.
- They are owned and managed by different, potentially mutually-distrustful organizations and individuals.

- They are potentially faulty.
- They have different security requirements and policies.
- They are heterogeneous, i.e., they have different CPU architectures, are running different operating systems, and have different amounts of memory and disk.
- They are connected by heterogeneous, multilevel networks.
- They have different resource management policies.
- They are likely to be geographically-separated (on a campus, in an enterprise, on a continent).

A Grid enables users to collaborate securely by sharing *processing, applications* and *data* across systems with the above characteristics in order to facilitate collaboration, faster application execution and easier access to data. More concretely this means being able to:

Find and share data. When users need access to data on other systems or networks, they should simply be able to access it like data on their own system. System boundaries that are not useful should be invisible to users who have been granted legitimate access to the information.

Find and share applications. The leading edge of development, engineering, and research efforts consists of custom applications – permanent or experimental, new or legacy, public-domain or proprietary. Each application has its own requirements. Why should application users have to jump through hoops to get applications together with the data sets needed for analysis?

Share computing resources. It sounds very simple – one group has computing cycles, some colleagues in another group need them. The first group should be able to grant access to its own computing power without compromising the rest of the network.

Grid computing is in many ways a novel way to construct applications. It has received a significant amount of recent press attention and been heralded as the next wave in computing. However, under the guises of "peer-to-peer systems", "metasystems" and "distributed systems", Grid computing requirements and the tools to meet these requirements have been under development for decades. Grid computing requirements address the issues that frequently confront a developer trying to construct applications for a grid. The novelty in grids is that these requirements are addressed by the grid infrastructure in order to reduce the burden on the application developer. The requirements (described more thoroughly in []) are: security, a global name space, fault-tolerance, accommodating heterogeneity, binary management, multi-language support, scalability, persistence, extensibility, site autonomy and complexity management.

Solving these requirements is the task of a Grid infrastructure. A architecture for a Grid based on sound design principles is required in order to address each of these requirements.

In this chapter we will focus on one particular aspect of grids – data. Data grids are used to provide secure access to remote data resources, flat-file data, relational data, and streaming data. For example, two collaborators at sites A and B need to share the results of a computation performed at site A, or perhaps design data for a new part needs to be accessible by multiple team members working on a new product at different sites – and in different companies.

We will examine in detail an Avaki Data Grid. An Avaki Data Grid provides transparent, secure, high-performance access to federated data sets across administrative domains and organizations. Users (both people and applications) of the Avaki Data Grid may be unaware that they are using a data grid. We begin with an examination of alternatives to data grid solutions. We then look at ADG in detail, starting with a close examination of the design principles and then the overall architecture. We follow with a look at performance.

Alternatives to Data Grids

The problems that data grids purport to 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 this file between Alice and Bob has been permitted.

Network File System – NFS

NFS is the classic 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 on to 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 accessed entirely on in parts, as desired. Finally, the NFS server and client tools come standard on all Unixes. On Windows, a special service pack must be purchased and installed.

The biggest disadvantage with NFS 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 unviable. 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 8KB. 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 clients do not.

Second, and most seriously, NFS does not address security well. An NFS request packet is sent in the clear and contains the (integer) UID and GID of the user making the read or write request. The NFS 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 trusted 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. Firewalls typically do not permit NFS traffic through them.

Third, assuming that the packets can be sent in a safe and trustworthy fashion, NFS requires that the identity spaces at the two sites to 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 is possible if Alice and Bob were within a single domain – in our realistic use-case, they are not.

There are other disadvantages plaguing 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, if Bob had some other 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.

File Transfer Protocol – ftp

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. Several of the commands resemble Unix commands, although several new commands, particularly for file transfer as well manipulating the local file system are different. 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's machine, 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 may be able to login into 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 inherently insecure – passwords are transmitted in the clear, as is data. 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, making ftp unviable. Even without firewalls, there are other disadvantages to using ftp. Since ftp requires making a copy of the data at Bob's machine, if Alice ever changes her own copy of the file, she must remember to ftp the new version of the file over. 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 the potential for inconsistencies. 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. Therefore, applications cannot take advantage of remote files using ftp without significant change.

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.

Secure copy – scp/sftp

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 is 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. Alice must continue to remember the appropriate machine name, username and password. She 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 keeping 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. Therefore, applications cannot take advantage of remote files using scp/sftp without significant change.

De-Militarized Zone – DMZ

A DMZ is simply a third set of machines accessible to both Alice and Bob using ftp or scp/sftp. 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 keeping three copies of the file. Also, because the file essentially makes two hops to get to its final destination, network usage is increased. DMZs do not ameliorate any of the other problems with scp/sftp. However, 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 remember yet another DMZ configuration for interacting with Chris. The same DMZ cannot be reused because of the potential for Chris to access files intended for Bob.

GridFTP

GridFTP is a tool for transferring files built on top of the Globus toolkit. 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 manner 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 accesses 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 consistency maintenance between multiple copies, because Alice and Bob would still require to keep at least two copies of the file, one on each user's machine.

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 scenario, 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. 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 is encrypted in transit. Authentication is using Kerberos. One drawback of using Kerberos is that the 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 can be a significant burden.

Avaki Data Grid

The objective of the Avaki Data Grid is to provide high-performance, easy, transparent, secure collaboration and coherent sharing between different administrative domains and organizations. Let's look briefly at each of these in turn.

High-performance. Nobody wants a low-performance system. Yet remote access is inherently slower than local access due to the combination of higher latency and often lower bandwidth. To provide high-performance access in the wide-area local cached copies must be made to reduce the time spent transferring data over the wide-area network.

Coherent. Caching data is great for performance. Unfortunately it can lead to inconsistent copies of the data, which can lead in turn to incorrect application results or bad decisions based on out-of-date data. Thus, the data grid must provide cache-coherent data while recognizing *and* exploiting the fact that different applications have different coherence requirements.

Easy and transparent. The data grid must be transparent to end users and applications. If users have to change their code or behaviors in order to use the data grid they are less likely to use it – reducing the benefit of having the grid in place.

Secure. "Secure" is a word that covers a whole range of issues. We believe that a data grid must support strong authentication with identities that span administrative domains and organizations, support the establishment of virtual organizations (groups that span organizations), enforce access control policies, and protect data.

Between different administrative domains and organizations. To span administrative domains a grid must address the identity mapping problem. To span organizations issues of trust management must be addressed.

In designing the Avaki Data Grid to meet these goals we kept three design principles in mind:

Provide a single-system view. With today's operating systems we can maintain the illusion that our local area network is a single computing resource. But once we move beyond the local network or cluster to a geographically-dispersed group of sites, perhaps consisting of several different types of platforms, the illusion breaks down. Researchers, engineers, and product development specialists (most of whom do not want to be experts in computer technology) must request access through the appropriate gatekeepers, manage multiple passwords, remember multiple protocols for interaction, keep track of where everything is located, and be aware of specific platform-dependent limitations (e.g., this file is too big to copy, or to transfer to one's system; that application runs only on a certain type of computer). Re-creating the illusion of single resource for heterogeneous, distributed resources reduces the complexity of the overall system and provides a single namespace.

Provide transparency as a means of hiding detail. Grid systems should support the traditional distributed system transparencies: access, location, heterogeneity, failure, migration, replication, scaling, concurrency, and behavior. For example, a user or programmer should not have to know where a peer object is located in order to use it (access, location, and migration transparency), nor does a user want to know that a component across the country failed – they want the system to recover automatically and complete the desired task (failure transparency). This is the traditional way to mask various aspects of the underlying system.

Reduce "activation energy". One of the typical problems in technology adoption is getting users to use it. If it is difficult to shift to a new technology then users will tend not to take the effort to try it unless their need is immediate and extremely compelling. This is not a problem unique to Grids – it is human nature. Therefore, one of our most important goals is to make using the technology easy. Using an analogy from chemistry, we keep the activation energy of adoption as low as possible. Thus, users can easily and readily realize the benefit of using Grids – and get the reaction going – creating a self-sustaining spread of Grid usage throughout the organization. This principle manifests itself in features such as "no recompilation" for applications to be ported to a Grid, and support for mapping a Grid to a local operating system file system. Another variant of this concept is the motto "no play, no pay". The basic idea is that if you do not need a feature, e.g., encrypted data streams, fault resilient files, or strong access control, you should not have to pay the overhead of using it.

The Avaki Data Grid meets our goals using a federated sharing model, a global name space, and a set of servers – called DGAS (Data Grid Access Servers) that support the NFS protocols and can be mounted by user machines – effectively mapping the data grid into the local file system. So, that is a lot of words. Let's break it down.

Let's start with the global name space. This is really a fancy word for a globally visible directory structure where the leaves may be files, directories, servers, users, groups, or any other named entity in the Avaki Data Grid. Thus, the path "/shares/grimshaw/myfile" names *myfile*, and the path can be used anywhere in the data grid by a client to refer to *myfile* regardless of where the client is located, and regardless of where *myfile* is located – or wherever *myfile* may migrate. There's no migration in ADG 3.0.

Data gets into the data grid – and gets a path name in the global name space when it is *shared*. The share command takes a rooted directory tree on some source machine and maps it into the global name space. For example, I can share c:\data on my laptop into /shares/grimshaw/data. At that point, the data on my laptop \data directory is available, subject to access control, by any authorized user in the grid for both read and write.

Data access in the ADG is, for the most part, via the local file system on the users machine, which in turn is an NFS client to a DGAS. The DGAS looks to the local operating system file system like a standard NFS 3.0 server. Thus the end user is unaware that they are even using the ADG; their shell scripts, Perl scripts, and other applications that use *stdio* will work without any modification on the ADG. We choose NFS because just about every operating system under the sun has a native NFS client.

Access control in ADG is via access control lists (ACL's) on each grid object (file, directory, share, group, etc.). For each operation on an object (e.g., read, write) there is an allow list and a deny list. These lists may contain both individual identities and group identities. A group is simply a set of individual identities. Groups are our mechanism for creating virtual organizations that span multiple actual organizations, and may contain identities from multiple organizations (grids). The allow list is first evaluated. If a request is not from a user in the allow list, the operation is rejected. Then, if the user is allowed, the deny list is checked. The deny list over-rides the allow list.

In the example in Figure 1 we have shared three different directory structures into the grid: one each from a Solaris machine, a WindowsNT machine and a Linux machine. The ADG does not care where the data is stored, on direct attached disk as in my example above, on NAS, on a SAN, or perhaps even on optical media. Furthermore, the data stays where it is. That means that local applications that count on the data in a particular directory can still access the data, and all local backup procedures continue to function.

Modifications to shared data by either direct means on the host machine, or via the ADG, are visible to both, though in the case of subsequent ADG access the coherence window applies (more on this later.)

Given the above example let's examine how the ADG is viewed by both the end user and by the system administrator.

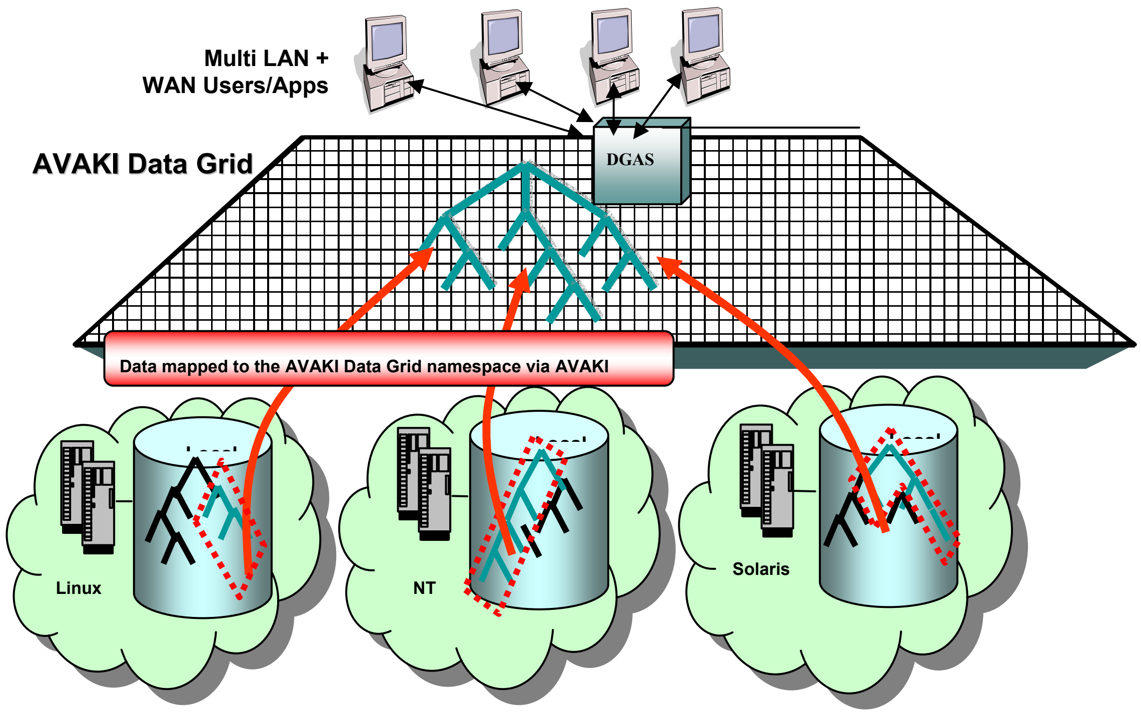


Figure 1. Data at three different sites, on three different types of machine have been mapped into the Avaki Data Grid. Data can now be accessed from anywhere in the Grid. Typically access is via a Data Grid Access Server (DGAS). The DGAS appears to local operating systems as an NFS 3.0 server, providing standardized, secure and transparent access to data.

User view

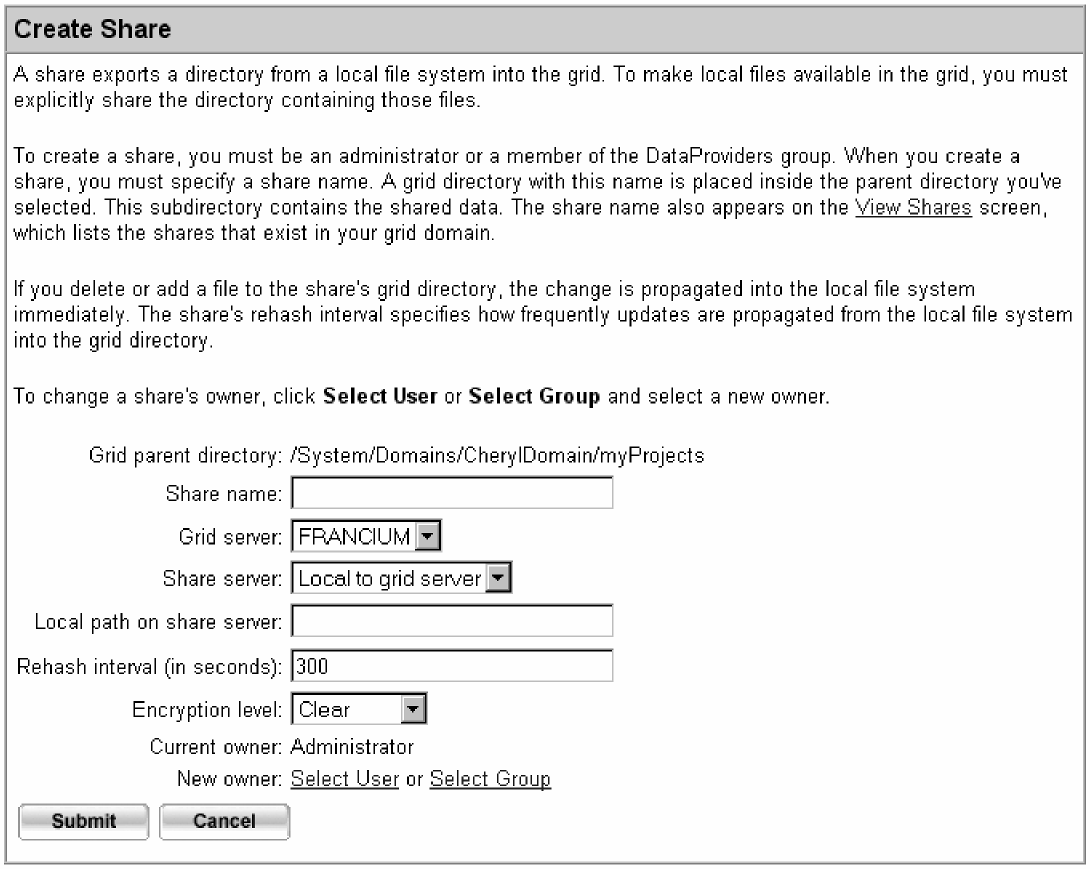
The first thing to stress about the user's view of the ADG is that no programming is required at all. Applications that access the local file system will work out of the box with the ADG. This is consistent with our goal of reducing the “activation energy” of grid adoption.

There are three ways for end users to access data in the data grid, via the local file system and an NFS mount of a DGAS, via a set of command lines tools, and via a web interface. In addition users may want to share some of their data into the grid, manage access control lists for files and directories that they own, etc.

NFS. We have already discussed access via the native file system. Applications require no modification, tools such as “ls” in Unix and “dir” in Windows work on mounted Avaki Data Grids. The user is not aware they are even using the ADG. A similar capability via CIFS will be available in the summer of 2003.

Command Line Interface. An Avaki Data Grid can be accessed using a set of command line tools that mimic the Unix file system commands such as *ls*, *cat*, etc. The Avaki analogues are *avaki ls*, *avaki cat*, etc. The Unix-like syntax is intended to mask the complexity of remote data access by presenting familiar semantics to users. The command line access tools are rarely used and are provided for users who may be unable for whatever reason to mount a DGAS into their file system.

Web-Based Portal. The third access mechanism is via a web-based portal. Using the portal user can traverse the directory structure, manage access control lists for files and directories, and create and remove shares. For example, in Figure 2 the interface to create a new share and map it into the ADG is shown.


Create Share

A share exports a directory from a local file system into the grid. To make local files available in the grid, you must explicitly share the directory containing those files.

To create a share, you must be an administrator or a member of the DataProviders group. When you create a share, you must specify a share name. A grid directory with this name is placed inside the parent directory you've selected. This subdirectory contains the shared data. The share name also appears on the View Shares screen, which lists the shares that exist in your grid domain.

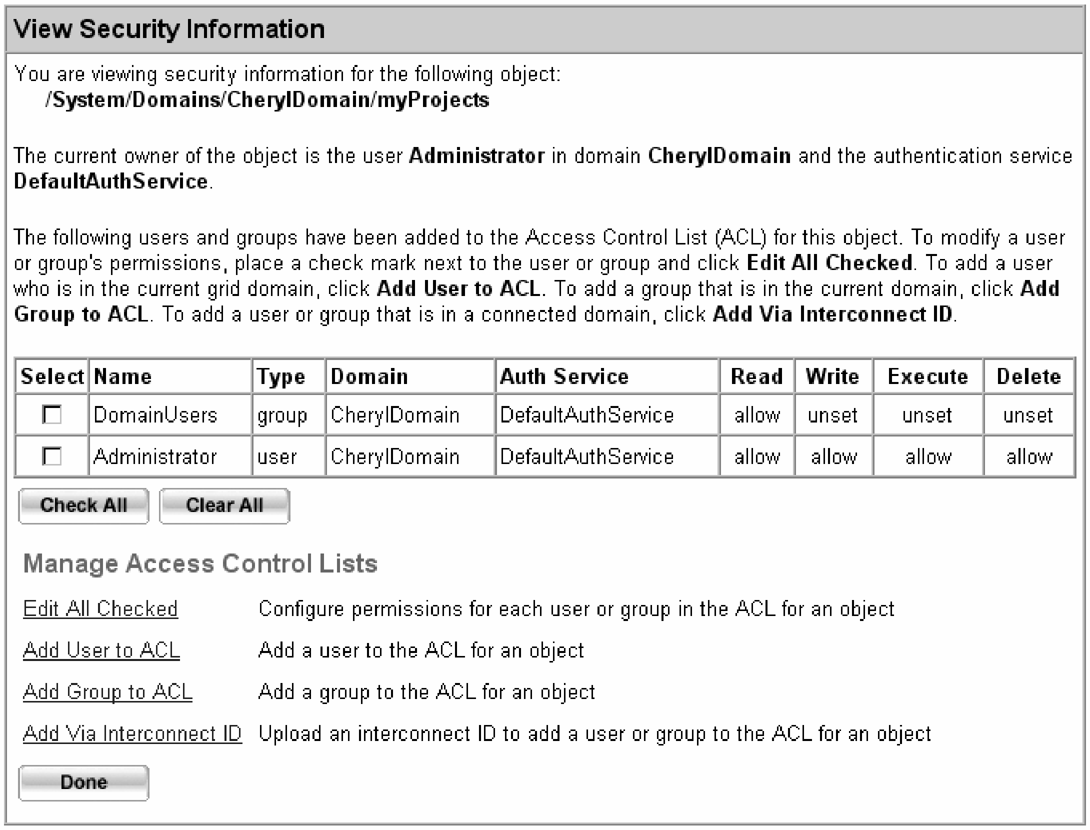
If you delete or add a file to the share's grid directory, the change is propagated into the local file system immediately. The share's rehash interval specifies how frequently updates are propagated from the local file system into the grid directory.

To change a share's owner, click **Select User** or **Select Group** and select a new owner.

Grid parent directory:	/System/Domains/CherylDomain/myProjects
Share name:	
Grid server:	FRANCIUM
Share server:	Local to grid server
Local path on share server:	
Rehash interval (in seconds):	300
Encryption level:	Clear
Current owner:	Administrator
New owner:	Select User or Select Group

Figure 2. The user uses the web-based GUI to add a share and thus map the data into the global directory structure. The user provides the name of the share, i.e., the path name, the local path of the data, the rehash interval, and the encryption level. In order to share a directory the user must be a member of the “DataProviders” group. Thus, management may restrict who is allowed to make data available on the grid.

The user can also use the web-based portal to view and modify access control lists on files, directories, groups. etc. For example, Figure 3 shows the ACL for an object “myProjects”. Figure 4 illustrates changing the ACL’s for an object “fred”.



View Security Information

You are viewing security information for the following object:
/System/Domains/CherylDomain/myProjects

The current owner of the object is the user **Administrator** in domain **CherylDomain** and the authentication service **DefaultAuthService**.

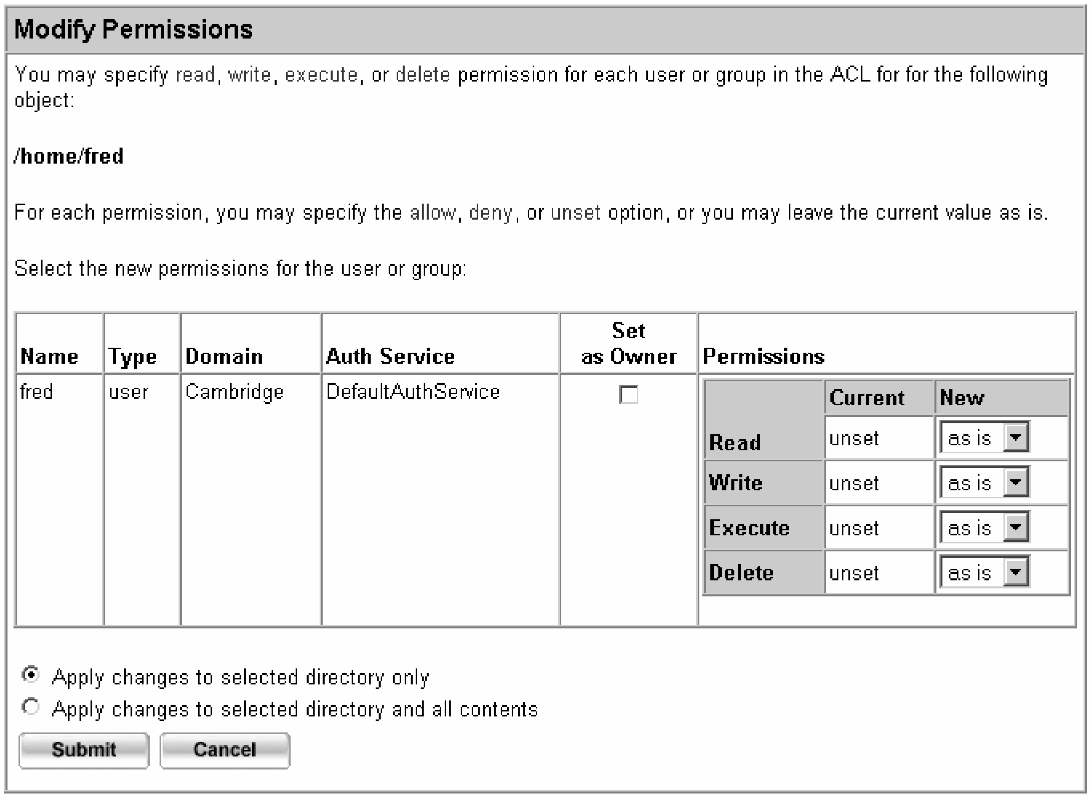
The following users and groups have been added to the Access Control List (ACL) for this object. To modify a user or group's permissions, place a check mark next to the user or group and click **Edit All Checked**. To add a user who is in the current grid domain, click **Add User to ACL**. To add a group that is in the current domain, click **Add Group to ACL**. To add a user or group that is in a connected domain, click **Add Via Interconnect ID**.

Select	Name	Type	Domain	Auth Service	Read	Write	Execute	Delete
☐	DomainUsers	group	CherylDomain	DefaultAuthService	allow	unset	unset	unset
☐	Administrator	user	CherylDomain	DefaultAuthService	allow	allow	allow	allow

Manage Access Control Lists

Edit All Checked	Configure permissions for each user or group in the ACL for an object
Add User to ACL	Add a user to the ACL for an object
Add Group to ACL	Add a group to the ACL for an object
Add Via Interconnect ID	Upload an interconnect ID to add a user or group to the ACL for an object

Figure 3. View/modify access control lists for an object. Both users and groups can be added to the access control lists via the links at the bottom of the page.



Modify Permissions

You may specify read, write, execute, or delete permission for each user or group in the ACL for for the following object:

/home/fred

For each permission, you may specify the allow, deny, or unset option, or you may leave the current value as is.

Select the new permissions for the user or group:

Name	Type	Domain	Auth Service	Set as Owner	Permissions
fred	user	Cambridge	DefaultAuthService	☐	

	Current	New
Read	unset	as is
Write	unset	as is
Execute	unset	as is
Delete	unset	as is

● Apply changes to selected directory only
○ Apply changes to selected directory and all contents

Figure 4. The access control lists for the object "/home/fred" is being modified. If it is a directory the changes can be applied recursively.

IT Manager View

Avaki ensures secure access to resources on the Grid. Files on participating computers become part of the Grid only when they are *shared*, or explicitly made available to the Grid. Further, even when shared, Avaki's fine-grain access control is used to prevent unauthorized access. Any subset of resources can be shared, for example, only certain files or directories. Resources that have not been shared are not visible to Grid users. By the same token, a user of an individual computer or network that participates in the Grid is not automatically a Grid user, and does not automatically have access to Grid files. Only users who have explicitly been granted access can take advantage of the shared resources. Local administrators may retain control over who can use their computers, at what time of day, and under which load conditions. Local resource owners control access to their resources.

An Avaki Grid can be administered in different ways, depending on the needs of the organization.

1. **As a single administrative domain**. When all resources on the Grid are owned or controlled by a single department or division, it is sometimes convenient to administer them centrally. The administrator controls which resources are made available to the Grid and grants access to resources. In this case, there may still be separate administrators at the different sites who are responsible for routine maintenance of the local systems.

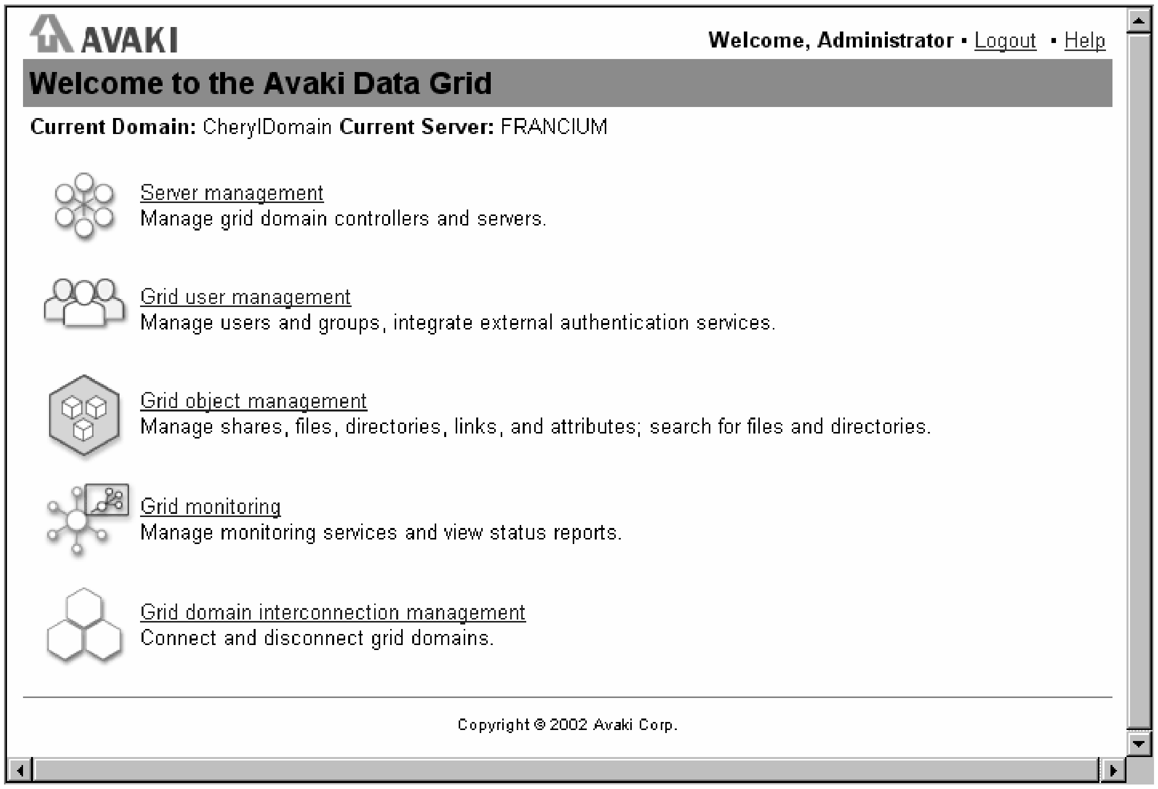
2. **As a federation of multiple grids – a grid of grids.** When resources are part of multiple administrative domains, as is the case with multiple divisions or companies cooperating on a project, more control is left to administrators of the local networks. They each define which of their resources are made available to the Grid and who has access. In this case, a team responsible for the collaboration would provide any necessary information to the system administrators, and would be responsible for the initial establishment of the Grid.

With Avaki, there is little or no intrinsic need for central administration of a Grid. Resource owners are administrators for their own resources and can define who has access to them. Initially administrators cooperate in order to create the Grid; after that, it is a simple matter of which management controls the organization wants to put in place.

Given the above, systems administrators perform basic tasks as shown in the system administrators main menu web page (Figure 5):

- Server management, where the number of grid servers is specified, and hot spares for high availability are configured.
- Grid user management, where users and groups are either imported from the existing LDAP, Active Directory, or NIS environment, or they are defined within the grid itself.
- Grid object management, where files and directories can be created and destroyed, ACL set, and new shares added.
- Grid monitoring, where logging levels, event triggers, and so on are set. ADG can be configured to generate SNMP traps and thus be integrated into the existing network management infrastructure.
- Grid interconnects, where the system administrator manages the set of grids to which this grid is connected.

In the following sections we will examine in more detail the architecture and servers that sit behind this interface. For more detail on system management options please see the Avaki ADG system administrators guide.


AVAKI

Welcome, Administrator • Logout • Help

Welcome to the Avaki Data Grid

Current Domain: CherylDomain **Current Server:** FRANCIUM

Server management
Manage grid domain controllers and servers.

Grid user management
Manage users and groups, integrate external authentication services.

Grid object management
Manage shares, files, directories, links, and attributes; search for files and directories.

Grid monitoring
Manage monitoring services and view status reports.

Grid domain interconnection management
Connect and disconnect grid domains.

Copyright © 2002 Avaki Corp.

Figure 5. The main menu for system administrators.

Architecture

ADG 3.0 has been written almost entirely in Java, with a small amount of native code primarily for performance. The architecture is based on off-the-shelf J2EE application servers. The application server currently being used is JBoss, although other application servers may be deployed with little or no code change.

Every ADG component runs within an application server. A Java application server is the equivalent of a traditional operating system but for J2EE components. Objects are created, deactivated, reactivated and destroyed within the application server on demand. Interactions between objects within the same application server are processed by the Java virtual machine within the application server. Interactions between objects in different application servers (typically, on different machines) are processed using remote method invocations (RMI). RMI calls are encrypted using Secure Sockets Layer (SSL) or any other off-the-shelf encryption technology. Interactions between objects may cause their internal states to be changed. The persistent state of objects is stored in an embedded database, currently either HyperSonic or CloudScape, accessed by the application server. All objects log several levels of messages using log4j, which stores these logs in files associated with the application server.

The major components of an ADG are: grid servers, share servers and data grid access servers (DGAS). A grid server performs grid operations such as authentication, access control and meta-data management. A share server performs bulk data transfer between a local disk on a machine and the grid. A data grid access server enables presenting the data grid as a Unix directory or a Windows drive using the NFS protocol. In this section, we discuss the components of an ADG. The interaction between these components provides an insight into the workings of a data grid.

Grid Servers

A grid server is the primary component of a grid. A grid server performs grid-related tasks such as domain creation, authentication, access control, meta-data management, monitoring, searching, etc. When deploying an ADG, the first grid server deployed typically bears the responsibility of starting a grid. This grid server is also called a grid domain controller (GDC). The GDC creates and defines a domain. A domain represents a single grid. Every domain has exactly one GDC. Multiple domains may be interconnected by invoking the appropriate functions on their respective GDCs.

A GDC is sufficient for creating, using and maintaining a small grid of 10-15 machines. However, beyond that number, maintaining scalability requires starting more grid servers on other machines. These grid servers are connected to the GDC. Each of these grid servers can be made responsible for a subset of the grid-related activities. For example, one of these grid servers can be made responsible for authentication. When a user logs into the grid, this grid server would be responsible for receiving the user name and password and either verifying the user's identity using an in-built grid authentication service or delegating this process to a third-party authentication service such as NIS, Active Directory or Netegrity. Once the user's identity has been verified, the grid server is responsible for generating credentials for this session for this user.

Another important task performed by a grid server is access control. When a user requests access to a file, directory or any other object in a grid, the grid server uses her credentials to retrieve her identity and then check the access controls on the object to determine if the requested access is permissible or not. Since the user may issue multiple requests when accessing a large file or when accessing a file repeatedly, the grid server may pass on the permission information in a handle to a share server in order to avoid repeated access control checks.

Yet another important task performed by a grid server is meta-data management. Every object in a grid has meta-data associated with it, such as creation time, ownership information, etc. For file objects, modification time and size are also meta-data. A grid server is responsible for storing this meta-data in an internal database, performing searches on it when requested and rehashing the information when it becomes stale.

A grid server can also be configured to perform monitoring services on other grid components. Monitoring typically involves determining the response time of other components to ping messages. As the ADG product evolves to incorporate database access, the grid server is expected to perform database tasks such as opening a connection, issuing a query or executing a stored procedure and reporting results into the data grid.

Share Servers

A share server is an ADG component that is responsible for bulk data transfer to and from a local disk on a machine. Multiple directories on a machine may be shared into the grid using a single share server; each of these directories is called a *share*. A share server is always associated with a grid server. The grid server is responsible for verifying whether a read/write request is permissible or not. If the request is permitted, the grid server passes a handle to the user as well as the share server. The user's request is then forwarded to the share server along with this handle. Subsequent requests are satisfied by the share server without the intervention of the grid server. Naturally, if the user issues a new request, for instance., to a new file, the grid server verifies the request anew before delegating the transfer to the share server.

A share server's main responsibility is to translate a grid read/write request into an equivalent read/write on the underlying file system. Depending on how the share server is configured, the translation may require decrypting data before writing to the file system and encrypting data after it has been read from the file system. Another responsibility of the share server is processing a rehash request initiated by its grid server. A rehash ensures consistency between the grid server's internal database about the contents of a share and the actual contents of the equivalent directory on the file system. Since sharing a directory does not preclude accessing the same directory using OS tools on that machine, it is possible for the contents of a share to be changed without any Avaki component being involved. A rehash restores the consistency of the data grid in such situations. Rehashes may be explicit or periodic.

A share server performs bulk data transfers, whereas its grid server performs grid-related tasks associated with the transfers. Incidentally, a grid server may also function as a share server, but not *vice versa*.

Data Grid Access Servers (DGAS)

A DGAS provides a standards-based mechanism to access an ADG. A DGAS is a server that responds to NFS 2.0/3.0 protocols and interacts with other data grid components. When an NFS client on a machine mounts a DGAS, it effectively maps the ADG global name space into the local file system, providing completely transparent access to data throughout the grid without even installing Avaki software. This NFS-based access to an ADG complements the command-line and web-based access that Avaki provides as part of every data grid deployment. Upcoming versions of the DGAS are expected to support the CIFS protocol for Windows clients as well.

Despite the functional similarity, a DGAS is not a typical NFS server. First, it has no actual disk or file system behind it; it interacts with components that may be distributed, be owned by multiple organizations, be behind firewalls, etc. Second, a DGAS supports the Avaki security mechanisms; access control is *via* signed credentials, and interactions with the data grid can be encrypted. Third, a DGAS caches data aggressively, using configurable local memory and disk caches to avoid wide-area network access. Furthermore, a DGAS can be modified to exploit semantic data that can be carried in the meta-data of a file object, such as "cacheable", "cacheable until" or "coherence window size". In effect, a DGAS provides a highly secure, wide-area NFS.

To avoid the rather obvious hot-spot of a single DGAS at each site, Avaki encourages deploying more than one DGAS per site. There are two extremes, one DGAS per site, and one DGAS per machine. Besides the obvious tradeoff between scalability and the shared cache effects of these two extremes, an added security benefit of having one DGAS per machine is that the DGAS can be configured to accept requests from only the local machine, eliminating the classic NFS security attacks *via* network spoofing.

Proxy Servers

A proxy server enables accesses across a firewall. A proxy server requires a single port in the firewall to be opened for TCP, specifically HTTP/HTTPS, traffic. All Avaki traffic passes through this port. Opening a firewall port essentially involves permitting traffic in and out of that port on the firewall machine and forwarding incoming traffic to another machine inside the firewall on which the Avaki proxy server is started. The proxy server accepts all Avaki traffic forwarded from the firewall and redirects the traffic to the appropriate components running on machines within the firewall. The responses of these machines are sent back to the proxy server, which forwards this traffic to the appropriate destination through the open port on the firewall.

A proxy server is associated with a grid domain "inside" a firewall. In other words, the proxy server and other grid servers and share servers must be in a common DNS domain and should be able to send messages to one another freely. Machines "outside" the firewall, i.e., in other DNS domains that are restricted by the firewall, must communicate with machines inside the firewall *via* the proxy server alone. The machines outside the firewall are not considered part the grid domain inside the firewall. Therefore, access through a firewall requires starting multiple grid domains (therefore, multiple GDCs) and then interconnecting them. Multiple grid domains may access a domain inside a firewall through the same proxy server. Two grid domains that are inside different firewalls may communicate with each other through one proxy server associated with each of them.

A proxy server may encrypt/decrypt as well as compress/uncompress data flowing through it. Message encryption maintains privacy and integrity of data grid traffic, whereas compression reduces network traffic, thus improving bandwidth. These operations occur transparently from the user's perspective as well as independent of the working of the rest of the grid components.

Failover Servers

A failover server is a grid server that serves as a backup for the GDC. A failover server is configured to synchronize its internal database periodically with a GDC. As a result, if a GDC becomes unavailable either because the machine on which it is running is down or because the network is partitioned or for any other reason, users can continue to access the grid without significant interruption in service. When a GDC is unavailable, all grid objects transparently access other grid objects using a failover server.

Grid objects access one another using a unique name, called a Location-independent Object IDentifier (LOID). The Avaki run-time system resolves LOIDs into location-specific identifiers encoded in Web Services Description Language (WSDL) documents. These WSDL documents typically include the address of the GDC. However, when a failover server is added to a grid, the address of the failover server is added to every WSDL. The run-time communication protocol for every object tries the addresses in the WSDL in order every time. If the first address, i.e., the GDC address is unreachable, the object automatically fails over to the next address, i.e., the first failover server. If even that address is unreachable, the object fails over to the address of the second failover server, if one is present, and so on until either an address is reachable or no addresses are found. In the latter case, the object reports an error and terminates the action. Thus, multiple failover servers may be configured for a single GDC, if desired.

The database within all failover servers is synchronized with the database within the GDC periodically. If a GDC becomes unavailable, the database in the next available failover server is guaranteed to be closely consistent with that of the GDC. However, subsequent actions may make the failover server database inconsistent with that of the currently-unavailable GDC. Therefore, when a grid is operating in failover mode, i.e., with a failover server acting *in lieu* of a GDC, actions that change the database are prohibited. Typically, this prohibition means that adding new shares or new files and directories to existing shares may be prohibited. Reading and writing existing files and directories can continue unhindered. This solution avoids some of the more difficult problems of fault-tolerance on a grid. For example, after a GDC becomes unavailable, failover servers do not have to vote among themselves to pick a new GDC – the next failover server listed in every WSDL automatically acts as a limited GDC. When the GDC returns, again no voting is required to pick the primary component of the grid – the GDC resumes that role simply because it continues to be the first address in every WSDL.

Performance

New Text

Performance is critical to acceptance of data grids. In particular, a frequent concern is the performance of a data grid relative to NFS, since NFS is the most commonly-deployed distributed file system. Accordingly, in this section we compare Avaki Data Grid performance against NFS. Data in an ADG can be accessed in many different ways; one of the ways makes the entire ADG look like an NFS-mounted directory. However, even in this for of access, different configurations of ADG components may give different performance results. In this study, we considered three configurations, which represent different use-case scenarios. We describe these ADG configurations, the NFS configuration used for comparison and the characteristics of the machines and tests.

Machine Configurations

We used three machines, testbed1, testbed9 and testbed17, for all of our tests. The characteristics of the machines are:

- testbed9:
    - Operating System: SunOS 5.8 Generic_108528-15 (Solaris 8)
    - Processor: sun4u sparc SUNW,UltraAX-i2
    - Memory: 512MB

  - Swap: 1.3GB
  - Local Disk: 6GB+
- testbed1:
  - Operating System: Linux 2.4.7-10 #1 (Red Hat 7.3)
  - Processor: i686
  - Memory: 512MB
  - Swap: 1.0GB
  - Local Disk: 25GB+
- testbed17:
  - Operating System: Linux 2.4.7-10 #1 (Red Hat 7.3)
  - Processor: i686
  - Memory: 512MB
  - Swap: 1.0GB
  - Local Disk: 8GB+
- Connectivity: 100Mb/sec

NFS Configuration

We used one of the machines as an NFS server and another as an NFS client:

- testbed9: NFS server
- testbed1: NFS client, mount parameters:
  - Protocol: TCP
  - Version: 3
  - Type: hard
  - Attribute Cache Timeout: 600 seconds

ADG Configuration

We used three configurations of the ADG, representing three use cases. The first configuration, called RemoteDGAS, places the Avaki DGAS on the same machine as the share server, but NFS client on a different machine. The second configuration, called LocalDGAS, places the Avaki DGAS on the same machine as the NFS client, but on a different machine from the share server. The third configuration, called WideDGAS, places the Avaki DGAS on a third machine, separate from the NFS client as well as the share server.

The RemoteDGAS configuration effectively compares the efficiency of Avaki components, i.e., the DGAS plus the share server plus its grid server, against that of a standard NFS server. In this configuration, the connection between the NFS client and the DGAS is the same as the connection between the same client and the standard NFS server.

The LocalDGAS configuration effectively compares the efficiency of the Avaki communication protocol against the NFS protocol.

The WideDGAS configuration presents a typical deployment configuration where the data to be shared is located on a different machine, possibly in a different DNS domain from the machine on which the data is to be accessed. This scenario cannot be duplicated with standard NFS.

RemoteDGAS

- testbed9: Grid Server (GDC), Share Server, DGAS
- testbed1: NFS client, mount parameters:
  - Protocol: TCP
  - Version: 3
  - Type: hard
  - Attribute Cache Timeout: 600 seconds

LocalDGAS

- testbed9: Grid Server (GDC), Share Server
- testbed1: DGAS, NFS client, mount parameters:
  - Protocol: TCP
  - Version: 3
  - Type: hard
  - Attribute Cache Timeout: 600 seconds

WideDGAS

- testbed9: Grid Server (GDC), Share Server
- testbed17: DGAS
- testbed1: NFS client, mount parameters:
  - Protocol: TCP
  - Version: 3
  - Type: hard
  - Attribute Cache Timeout: 600 seconds

For all configurations, we accessed two shares on the Share Server's local directory. One share was configured to provide data encrypted with SSL, while the other share was configured to provide data unencrypted.

Test Configuration

For the tests, we measured wall-clock time taken, with precision in microseconds, to perform write and read operations. Each write operation was performed five times. Each read operation was performed five times, recognizing that the first time represented an uncached access, whereas the remaining four times represented cached access.

We took care to issue umounts for the NFS clients between each write and read operation so as to eliminate the effects of caching at the mount client. This caching benefits both ADG and NFS equally. We took care to ensure that the DGAS caches were stored on local disks, not NFS-mounted directories to eliminate one source of delays within the DGAS. Also, the DGAS was configured to print information, but not debug messages. We ensured that the results of every operation were correct. For writes, we checked the file size as reported by the Unix tool “ls”. For reads, we added several checks, such as “wc”, “diff” and “sum”. We took care to ensure that the checks did not pollute cached read results. We performed each operation using the Unix command “cp” as well as using a Unix program which performed the same effective operation. For the command-based operation, we measured the time to complete the entire command. For the program-based operation, we measured the time to complete exactly the required data transfer, thus eliminating program startup and shutdown. For the program-based transfer, we kept the buffer block size at 8KB, which is a traditional standard for NFS. For wide-area ADG, we recommend changing this block size to a larger size, say 128KB, to amortize latency. Also, we kept program block size, i.e., the block size written by the program every single time at 128B.

In a second test, we performed 500000 random accesses on a 64MB file. Each random access was either a read, write or seek of a random amount of bytes. Again, we performed each operation 5 times. This test was meant to capture the performance under non-sequential access, such as those by an application. For this test, we plotted the time taken, not bandwidth.

Test Results

We expected NFS to outperform ADG. Since our tests were in a LAN environment, the overhead of using multiple servers in ADG as opposed to one server in NFS plays an important factor. In a WAN environment, the greater transmission latencies involved would mask these overheads. Besides, in a WAN environment, we have been unable to get NFS to finish all of the tests undertaken here successfully (results not shown here). Again, since NFS is not a WAN protocol, the failures are unsurprising.

Between the three ADG configurations, we did not notice significant performance difference. This lack of difference is explained by the LAN environment. An interesting area of further study is the performance difference of the three configurations in a WAN environment. While we can speculate on the expected relative performances, we will defer that exercise to a later study. Another unsurprising result of our study is that when shares are configured to serve data encrypted, performance drops. The surprise was in the amount of performance drop – as much as 100% in some cases. Program-based I/O *versus* command-based I/O did not result in significant differences, except for very small file size, where command startup and shutdown dominates actual transfer. Therefore, unless explicitly mentioned, all of the results shown here are for program-based I/O. These results are presented in Figures 6-10.

The plot in Figure 6 shows DGAS write performance for unencrypted shares compared to NFS using a program in all configurations. NFS write performance is consistently better than Avaki write performance, as is expected. For NFS vs. RemoteDGAS, Avaki requires passing traffic through twice as many servers as NFS. For NFS vs. LocalDGAS, Avaki traffic generally has more volume than NFS traffic because of grid headers and encryption. For NFS vs. WideDGAS, Avaki traffic makes two network hops as opposed to NFS traffic. In Figure 7, we show performance for uncached reads. In general, uncached reads are relatively similar in performance to writes for all configurations. As compared to writes, reads result in poorer performance because they are synchronous.

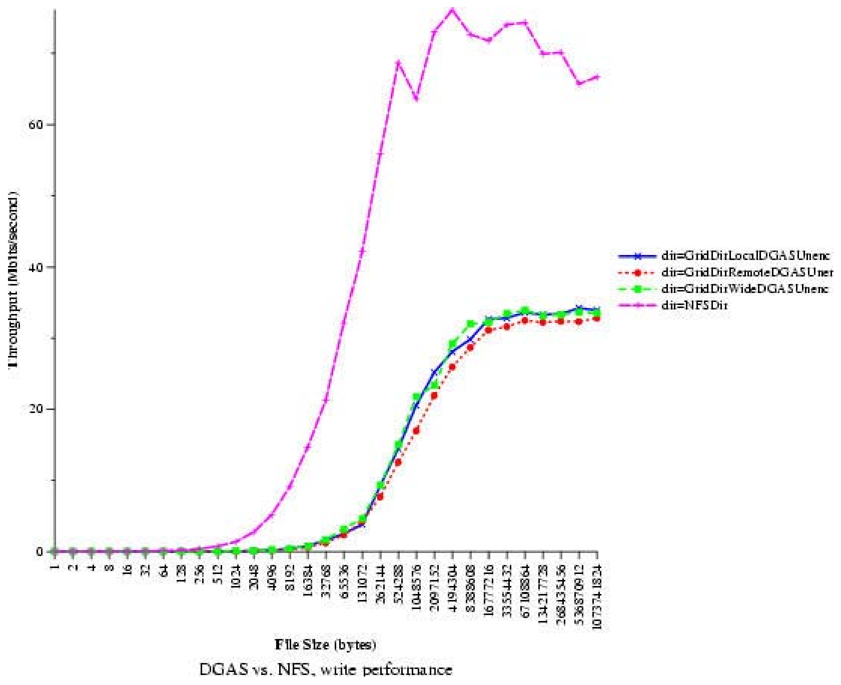


Figure 6. Write Performance comparison between ADG 3.0 and native NFS.

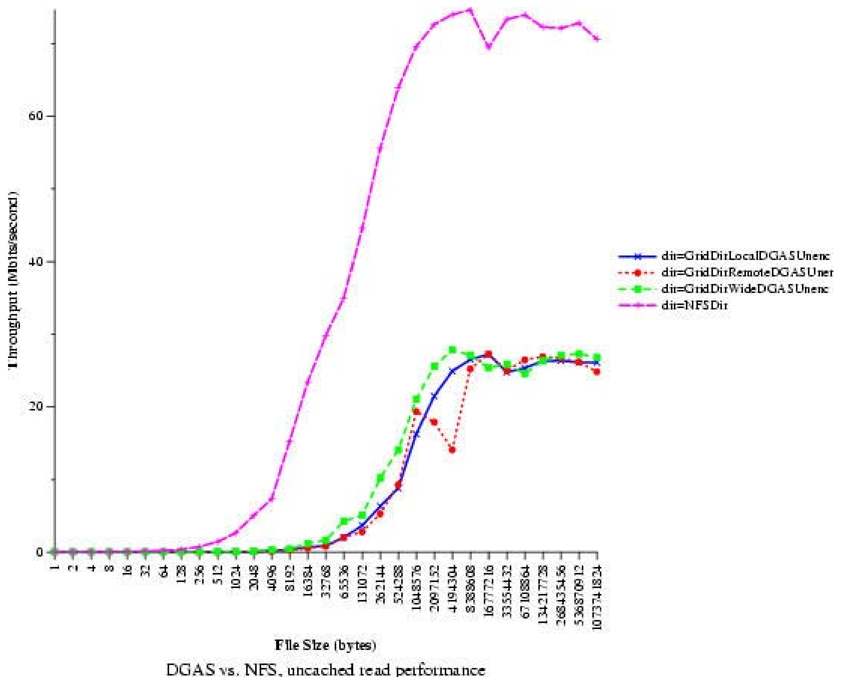


Figure 7. Uncached Read Performance comparison between ADG 3.0 and native NFS.

For cached reads, as shown in Figure 8, the DGAS cache results in better performance for all DGAS scenarios. When file size becomes bigger than the cache, naturally, performance drops because the cache can no longer satisfy the operation. Contention between the NFS client and DGAS in LocalDGAS and the DGAS and share server in Remote DGAS results in poorer performance than WideDGAS.

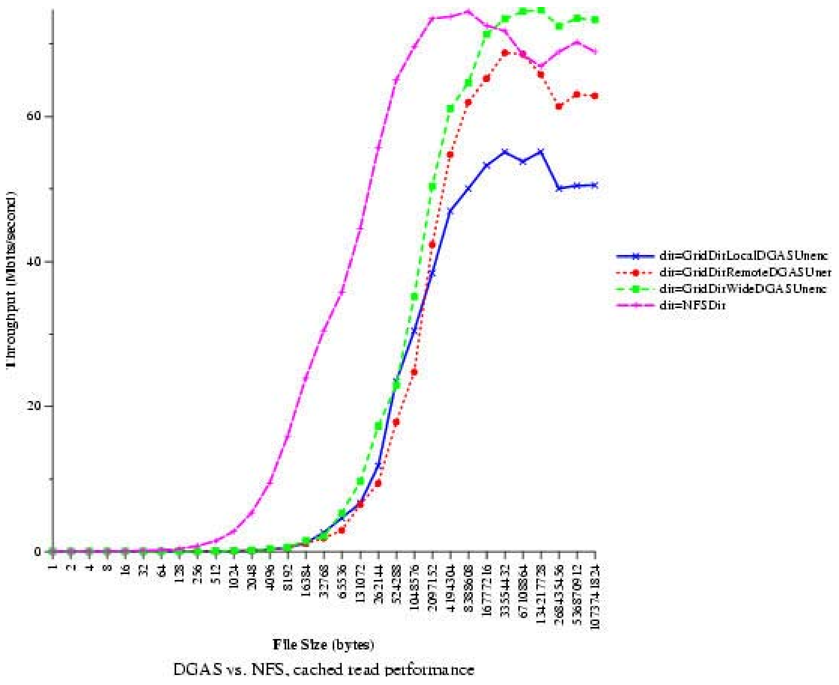


Figure 8. Cached Read Performance comparison between ADG 3.0 and native NFS.

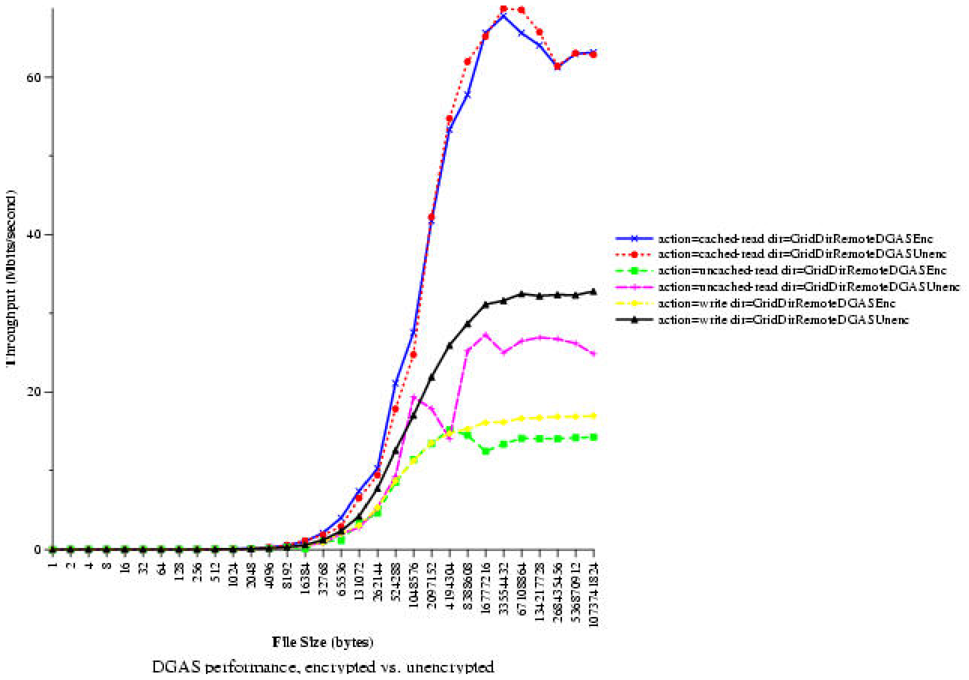


Figure 9. Encrypted/Unecrypted Read/Write Performance for ADG 3.0.

The plot in Figure 9 shows the effect of encrypting shares. The DGAS pays the penalty of encryption on writes and uncached reads, but not otherwise, since the cached copy is unencrypted. Encryption penalty can be as high as 100%. Incidentally, this plot also compares performance of writes, cached reads and uncached reads. Performance of writes and uncached reads are similar, but far inferior to performance of cached reads.

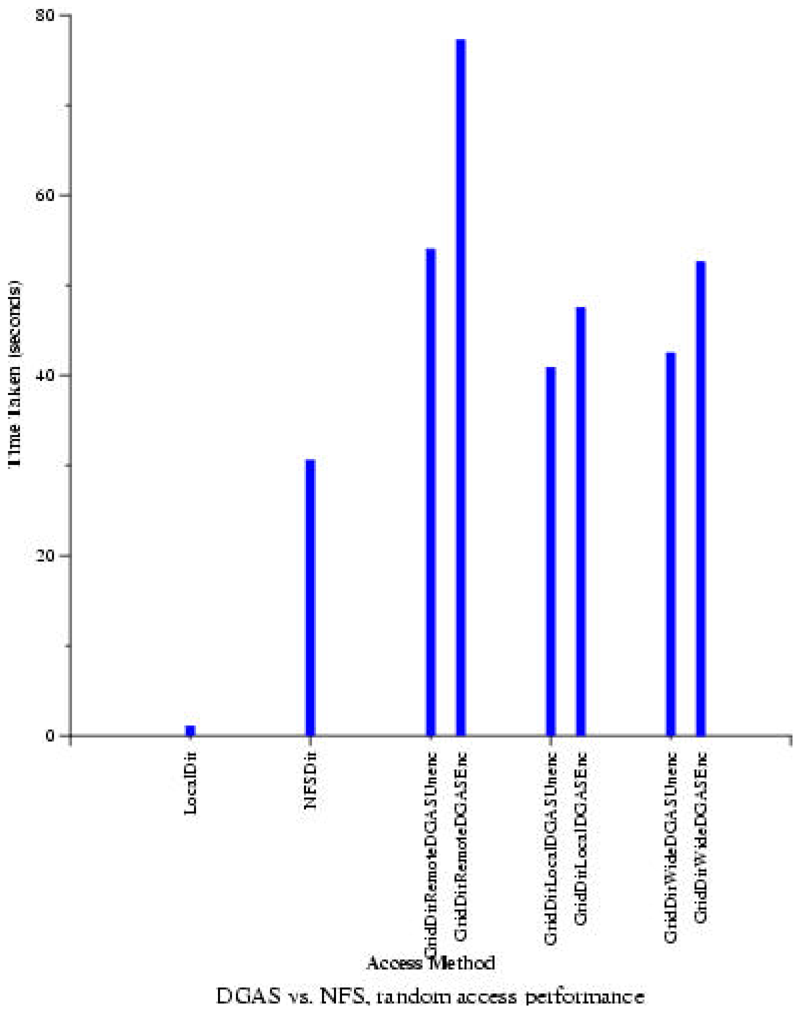


Figure 10. Random Access Performance comparison between ADG 3.0 and native NFS.

Random access performance of DGAS is about 50% worse than that of NFS for unencrypted shares. The performance for encrypted shares is worse, expectedly so.

Old Text

Performance is critical to acceptance of data grids. In particular how the data grid compares with the native NFS performance is a key factor – since NFS is the most frequently used distributed file system. Below we compare the performance of Avaki ADG 3.0 versus the native NFS on Linux in a local area, 100 mbs Ethernet environment.

The test environment for the native NFS consists of two 933 MHZ PIII's running Red Hat Linux 7.1. Each machine had 512 MB of memory. The NFS server was on one machine, and the client test application on another. Performance for ADG was measured with three machines. Like the native NFS the NFS client and test program ran on the same 933 MHZ PIII, and the DGAS ran on the other 933 MHZ PIII. The share server (where the data actually lives) was a 2.4 GHZ P4 with 1 GB of memory running Windows 2000.

Performance was measured for file sizes from 1KB to 1GB, incrementing by a factor of 2, e.g., 1K, 2K, 4K, 8K, and so on till 1 GB. I/O throughput was measured by starting a timer, performing Unix “cp”, stopping the timer, and computing the throughput. To eliminate local operating system cache effects the NFS server was unmounted between trials (for both native NFS and ADG).

The performance results are shown in Figure 11. A few things to note about the results. First, the cached ADG reads (client to DGAS read) performance is very similar to the native NFS read performance, particularly for larger files. Second, the ADG un-cached reads and ADG writes are approximately half the speed of NFS. The reason is the network and client configuration. Our client, DGAS, and true copy of the data are all on different machines, requiring two network hops rather than one. We choose this configuration (rather than placing for example the DGAS and the share on the same machine) because we believe it is more representative of actual deployments. Third, and most significantly, if we had run this same experiment with the native NFS server on a remote machine and the Avaki share server on a remote machine the Avaki cached read performance would be the same. The native NFS would be much worse. Indeed it can be hard to test because NFS often times out and fails on wide area systems.

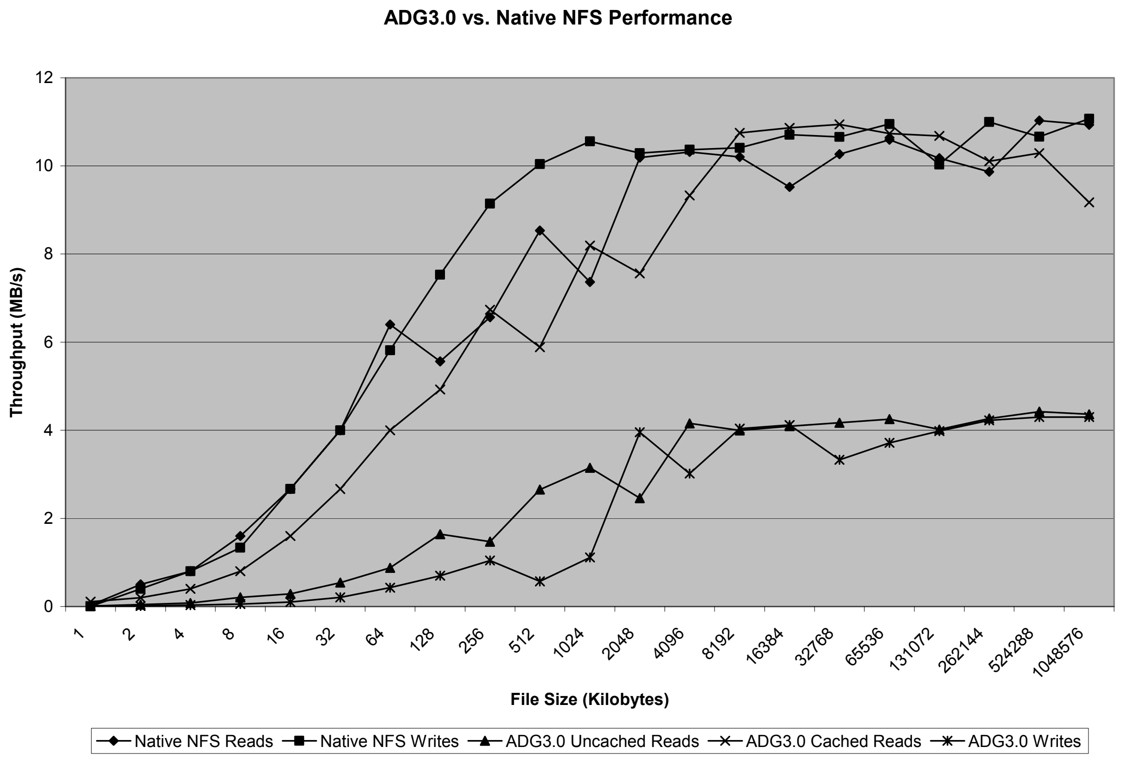


Figure 11. Performance comparison between Avaki ADG 3.0 and native NFS.

Summary

In this chapter we have presented the Avaki Data Grid, its usage, architecture, and performance. The performance of ADG was compared to native NFS. ADG performance was competitive with native NFS in the most common usage scenario – reading. Further we examined the alternatives to data grid technology – ftp, scp, wide area NFS, and DMZ's and found they fall short on several dimensions.

The bottom line is that Avaki provides high-performance, easy, transparent, secure collaboration and coherent sharing between different administrative domains and organizations. This allows organizations to reduce the "friction" of collaboration both internally and externally, reducing both costs and time to market.

References

[1] A.S. Grimshaw, "Enterprise-Wide Computing," *Science*, 256: 892-894, Aug 12, 1994.

[2] A.S. Grimshaw and W.A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, 40(1): 39-45, Jan 1997.

[3] L. Smarr and C.E. Catlett, "Metacomputing, *Communications of the ACM.* 35(6):44-52, June 1992.

[4] FTP specification, http://www.ietf.org/rfc/rfc0765.txt?number=765