FILE SYSTEM VIRTUALIZATION AND SERVICE FOR GRID DATA MANAGEMENT

By

MING ZHAO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF THE PHILOSOPHY

UNIVERSITY OF FLORIDA

2008

To my wife and my parents

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Renato Figueiredo, for his excellent guidance throughout my Ph.D. study. I am greatly indebted to him for providing me such an exciting research opportunity, constantly supporting me on things no matter big or small, and patiently giving me time and helping me grow. I would also like to gratefully and sincerely thank Prof. José Fortes for his exceptional leadership of the ACIS lab and his invaluable advice in every aspect of my academic pursuit. I have learned enormously from them about research, teaching, and advising, and they will always be examples for me to follow in my future career.

I am very grateful to the other members of my supervisory committee, Prof. Sanjay Ranka and Prof. Tao Li, for taking time out of their busy schedules to review my work. Their constructive criticism and comments are very helpful to improving my work and are highly appreciated. I also wish to thank Prof. Alan George and Prof. Oscar Boykin for their advice and help.

My heartfelt thanks and appreciation are extended to my current and former fellow ACIS lab members. The lab is where I have obtained solid support of my research, gained precious knowledge and experience, and grown from a student to a professional. I am especially thankful to my colleague and good friend, Prapaporn, for her careful proofing of the manuscript as well as our close collaboration in the DDDBMI project.

A special note of gratitude is due to Dr. Gang Rong at Tsinghua University, my formal advisor on my master's degree study, for encouraging and assisting me to pursue my Ph.D. overseas.

Finally, and most importantly, I would like to thank my family and I owe everything I have achieved to them. My parents' unwavering belief in me and unending caring of me are what have made me the person I am today. My brother, Hui, has been my best friend since my childhood and has always been there when I need him. My wife, Jing, has been both a loving companion and a supportive colleague, bringing endless inspiration, joy, and

passion into my life. It is truly wonderful to have her sharing every moment with me in the past five years, and I look forward to walking hand in hand with her through the new journey ahead of us.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of the Philosophy

FILE SYSTEM VIRTUALIZATION AND SERVICE FOR GRID DATA MANAGEMENT

By

Ming Zhao

August 2008

Chair: Renato J. Figueiredo
Major: Electrical and Computer Engineering

Large-scale distributed computing systems, such as computational grids, aggregate computing and storage resources from multiple organizations to foster collaborations and facilitate problem solving through shared access to large volumes of data and high-performance machines. Data management in these systems is particularly challenging because of the heterogeneity, dynamism, size, and distribution of such grid-style environments. This dissertation address these challenges with a two-level data management system, in which file system virtualization provides application-tailored grid-wide data access, and service-based middleware enables autonomic management of the data provisioning.

The diversity of applications and resources requires a data provisioning solution that can be transparently deployed, whereas the dynamic, wide-area environments necessitate tailored optimizations for data access. To achieve these goals, this dissertation proposes grid-wide virtual file systems (GVFS), a novel approach that virtualizes existing kernel distributed file systems (NFS) with user-level proxies, and provides transparent cross-domain data access to applications. User-level enhancements designed for grid-style environments are provided upon the virtualization layer in GVFS, including: customizable disk caching and multithreading for high-performance data access, efficient consistency protocols for application-desired data coherence, strong and grid-compatible security for secure grid-wide data access, and reliability protocols supporting application-transparent

failure detection and recovery. Based on GVFS, data sessions can be created on demand on a per-application basis, where each session can apply and configure these enhancements independently.

The second level of the proposed data management system addresses the problems of managing data provisioning in a large, dynamic system: how to control the data access for many applications based on their needs, and how to optimize it automatically according to high-level objectives. It proposes service-based middleware to manage the lifecycles and configurations of dynamic GVFS sessions. These data management services are able to exploit application knowledge to flexibly customize data sessions, and support interoperability with other middleware based on Web Service Resource Framework. In order to further reduce the complexity of managing data sessions and adapt them promptly to changing environments, an autonomic data management system is built by evolving these services into self-managing elements. Autonomic functions are integrated into the services to provide goal-driven automatic control of GVFS sessions on the aspects including cache configuration, data replication, and session redirection.

A prototype of the proposed system is evaluated with a series of experiments based on file system benchmarks and typical grid applications. The results demonstrate that GVFS can transparently enable on-demand grid-wide data access with application-tailored enhancements; the proposed enhancements can achieve strong cache consistency, security, and reliability, as well as substantially outperform traditional DFS approaches (NFS) in wide-area networks; the autonomic services support flexible and dynamic management of GVFS sessions, and can also automatically optimize them on performance and reliability in the presence of changing resource availability.

CHAPTER 1
INTRODUCTION

Computations are becoming increasingly larger scale, in terms of both size and geographical and administration distribution. Examples include scientific grids [1] which harness resources among several institutions for coordinated problem solving, and enterprise information systems that aggregate efforts from multiple sites for collaborative development. Common in these systems is that applications and data are distributed on resources across administrative boundaries and wide-area networks. Such environments can be referred as the *"grid-style" environments*, which have the following distinctive characteristics:

- **Heterogeneity**: There exist a wide variety of applications and resources in a grid-style environment. The resources typically have different hardware configurations (e.g., CPU speed and architecture, memory size, disk bandwidth and capacity) and software setups (e.g., operating systems and libraries); the applications also have diverse characteristics (e.g., data access pattern) and needs (e.g., desired data access performance, security, and reliability).

- **Dynamism**: Systems deployed in a grid-style environment are highly dynamic. Failures on machines and networks can happen at any time, and non-dedicated resources may dynamically join and leave the system. On the other hand, applications are started and terminated on demand, and their workloads also vary over time.

- **Scale**: Large amounts of resources can be aggregated in a grid-style environment. They are distributed across different institutions and connected on wide-area networks, providing the computing power and storage capacity to support executions of many applications.

This dissertation focuses on two specific aspects of data management in distributed systems: *data provisioning* — providing applications running on the computing resources with remote access to their data stored on the storage resources, and the *management of the data provisioning* — the establishment, configuration, and termination of the remote data access. Computing in a grid-style environment poses unique challenges to these tasks because of the above mentioned heterogeneous, dynamic, and large-scale nature of applications and resources.

First, the diversity of applications and resources motivates a data provisioning solution that can be transparently deployed, without modifying the existing operating systems (O/Ss) and changing the application source code or binaries. Second, the wide-area, cross-domain environments necessitate application-tailored optimizations for data access to address the inefficiency (long network delay, limited network bandwidth), insecurity (insecure resources, limited mutual-trust between different domains), and unsafety (unreliable machines and networks) that are typical in such environments. Last but not least, the management of data provisioning in a large, dynamic system also desires flexible control and automatic optimization of the remote data access, in order to deal with the complexity of providing data to many applications, to agilely adapt to the changing environments, and to deliver application-desired performance, security, and reliability.

To address these challenges, this dissertation presents a two-level data management system in which file system virtualization provides application-tailored grid-wide data access, and service-based middleware enables autonomic management of the data provisioning. In particular, this system has made the following contributions:

- It provides on-demand, cross-domain data access transparently for unmodified applications and O/Ss based on user-level virtualization of widely available O/S-level distributed file systems (DFSs).

- It supports application-tailored enhancements designed for grid-style environments on several important aspects of remote data access, including performance, consistency, security, and reliability.

- It employs middleware services to achieve flexible and interoperable management of grid-scale data provisioning, which is capable of controlling the lifecycles and configurations of dynamic data sessions based on application needs.

- It develops autonomic functions to automatically optimize the data management according to high-level objectives, in order to reduce the complexity of managing data sessions and adapt them promptly to changing environments.

- Finally, the proposed system has been demonstrated, with thorough experimental evaluation, that it is effective and can significantly outperform conventional

DFS-based approaches in grid-style environments; it has also been successfully deployed in a production grid system [2][3] for several years, supporting scientific tools and users from many disciplines.

The data management system proposed in this dissertation is architected to address three important questions, which are discussed in the following subsections respectively.

1.1 Application-Transparent Grid-Wide Data Access

The first question is, how to provide application-transparent grid-wide data access?

Grids differ from traditional distributed computing environments because of their distinct characteristics, e.g., wide-area networking, heterogeneous end systems, and disjoint administrative domains. These differences bring new challenges to data management systems, and the technologies that are successful in local-area networks (LAN), e.g., LAN file systems, cannot be directly applied in a grid environment. Instead, grid data management needs to specifically address these unique issues.

Existing solutions allow applications to access grid data through the use of specialized APIs or libraries. However, the required modifications on application sources or binaries often place a burden upon the shoulders of end users and developers, and present a hurdle to applications that cannot be easily modified. Therefore, application-transparency is desirable to facilitate the deployment of a wide range of applications on grids, where grid-enabling should be the responsibility of the grid middleware but not the application users or developers.

This dissertation presents a user-level DFS virtualization, namely **G**rid **V**irtual **F**ile **S**ystem (GVFS), for application-transparent grid data access. Because the well-known DFS interface is preserved by GVFS and presented to applications, no modifications are required to their source code, libraries, or binaries. In addition, the proposed approach is based on user-level virtualization techniques, which requires no changes to existing O/Ss and can be conveniently deployed on grid resources. Furthermore, user-level enhancements designed for grid-style environments are built upon the virtualization layer to enable data provisioning with application-desired characteristics.

In short, the proposed GVFS approach answers the first question by providing transparent grid-wide data access for unmodified applications and O/Ss through the user-level DFS virtualization.

1.2 Application-Tailored Grid Data Provisioning

The second question is, how to provide data with application-tailored optimizations?

Typical O/Ss are designed to support general-purpose applications, but it is often the case that "one size does not fit all". Applications have diverse characteristics and requirements, in terms of, for example, data access patterns, acceptable caching and consistency policies, security concerns, and fault tolerance requirements. To provide the desired performance, security, and reliability to a grid application, data provisioning needs to be optimized according to the application's behaviors and needs.

Because an optimization tailored for one application (e.g., aggressive prefetching of file contents) may result in performance degradation for several others (e.g., sparse files, databases), application-tailored features are typically not implemented in general-purpose O/S kernels. In addition, kernel-level modifications are difficult to port and deploy, notably in shared environments. Toolkit-based solutions typically give users powerful APIs to program remote data access with desired behaviors, but few programmers are skilled to make effective use of such APIs.

To solve this problem, user-level DFS customizations are proposed to support application-tailored GVFS data sessions. In particular, enhancements designed for grid-style environments are provided upon the virtualization layer in GVFS, which include customizable disk caching and multithreading for high-performance data access, efficient consistency protocols for application-desired data coherence, strong and grid-compatible security for secure grid-wide data access, and reliability protocols supporting application-transparent failure detection and recovery. Based on GVFS, data sessions can be created on demand on a per-application basis, where each session can apply and configure these enhancements independently to address its application's needs.

Therefore, the answer to the second question is to use the application-tailored enhancements enabled by GVFS to provide grid-wide data sessions with application-desired performance, consistency, security, and reliability.

1.3 Service-Based Autonomic Data Management

The third question is, how to manage data provisioning in a grid-scale system with dynamically changing environments?

Based on the GVFS approach, data sessions can be started on demand and independently customized for applications. However, in a large-scale system, the management of many dynamic data sessions is another challenging task due to its complexity. Data sessions need to be dynamically established and destroyed based on the lifecycles of applications and the locations of their instantiations and data storage. Customization of data sessions also implies the consideration of various relevant factors and tuning of many parameters, in accordance with the desired behaviors and the surrounding environments. Dynamically changing application workload and resource availability further require continuous monitoring of data sessions and timely adaptation of their configurations.

These requirements are often beyond the capability of end-users and even system administrators. Yet the goals of users or administrators are rather simple and explicit. For example, from an application user's point of view, it is desired that the job execution is fast, secure, and reliable; from a resource provider's point of view, it is expected that the resource use is healthy and profitable. Therefore, this dissertation presents a novel service-based autonomic data management approach to automatically manage and optimize the data provisioning according to such high-level objectives.

This dissertation proposes a set of data management services to manage the per-application GVFS sessions, enforce the isolation among the independent sessions, and apply the desired customization for each session. They support flexible control over the lifecycles and configurations of data sessions, and can explore the knowledge

of applications (e.g., data access patterns, data sharing scenarios, and service quality requirements) to customize their data sessions on the use of performance, consistency, security, and reliability enhancements. These services also provide interoperable interfaces which allow for direct interactions with other grid middleware services and automated executions of data provisioning tasks.

To further reduce human intervention in managing data sessions and enable them to promptly adapt to the changing environments, autonomic functions are built into the data management services to make them capable of automatically monitoring, analyzing, and optimizing the distributed entities of grid-wide data sessions, and cooperatively working together to achieve the desired data provisioning and resource usage goals. Such autonomic management is applied to several important aspects of data sessions including cache configuration, data replication, and session redirection.

In summary, the GVFS-based data management system addresses the last question by employing autonomic services to provide automatic management and optimization of data sessions according to the application needs and changing environments.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Typical Grid Data Management Approaches

Currently there are three main approaches to grid data provisioning, which are summarized as follows:

The first approach leverages middleware to explicitly transfer files prior to and after application execution. This approach is often called "file staging" and is adopted by several major cluster computing and grid computing systems, such as PBS [4] and Globus [5]. Typically, the necessary inputs are staged in before a job starts and the produced outputs are staged out after the job completes, where the files are transferred entirely using tools such as RCP (Remote Copy), SCP (Secure Copy), and GridFTP [6].

The second approach is based on application programming interfaces (APIs) which allow an application to explicitly control transfers. For example, Globus GridFTP [6] and GASS [7] provide APIs for applications to download and upload files entirely for access, whereas the RFT (Reliable File Transfer [8]) service exposes Web service based interface for scheduling GridFTP-based file transfers.

The third approach employs mechanisms to intercept data-related events and handle them with remote data access implicitly to the applications. For example, standard C library calls (e.g., fread, fwrite) and Linux system calls (e.g., read, write) for local file access can be intercepted and mapped to operations on a remote file [9][10][11][12]; distributed file system (DFS) calls (e.g., NFS read and write remote procedure calls) on a regular file can be mapped to the access on a grid object [13][14].

Comparing these three different approaches, the first one is traditionally taken for applications with well-defined data sets and flows, such as uploading of standard input and downloading of standard output. It is difficult to support applications that have obscure, complex data access patterns. The second approach is adopted for applications where the development cost of incorporating specialized APIs is justifiable for certain

specific purposes, such as performance and security. It is not applicable for applications that do not have source code available, such as packaged commercial software. The efforts required for the modifications also make it difficult to port a wide variety of applications. In addition, both of the first two approaches need to transfer files in their entirety in order to read or write them. Thus, they are not efficient for files that are only accessed sparsely (e.g., database files, virtual machine disk state) and they cannot support fine-grained data sharing among several distributed users or applications.

In contrast, the third approach achieves great application transparency and can be used for applications that do not have well-defined data sets or access patterns and for applications that cannot be easily modified. It is also possible for this approach to transfer only the needed data blocks of files on demand, so that sparse data access on large files can be efficient and multiple users or applications can flexibly work on the same files concurrently.

The **G**rid **V**irtual **F**ile **S**ystem (GVFS) described in this dissertation is based on the third approach. Experience with network-computing environments has shown that there are many applications in need of such solutions [2][15]. The Condor [9] and Kangaroo [16] systems provide remote data access to an application through library call interception by means of either static or dynamic relinking. Static linking requires the application to be linked to a specialized library that replaces the existing one (e.g., the standard C library), so that the data-related library calls can be intercepted and mapped to remote I/O operations [9]. Dynamic linking achieves the same goal by using linker control to direct the specialized dynamic library to be used in place of the existing one when the application is executed. However, static linking requires rebuilding the application and does not work if its source code is not available; dynamic linking only works for applications that are dynamically linked and does not support statically-linked applications. In addition, Kangaroo does not provide full file system semantics and thus

cannot support many applications that require these missing operations (e.g., delete and link).

Previous effort on the UFO system [11] and recently, the Parrot file system [12], leverage system call tracing to intercept an application's data-related system calls and map them to remote data access for the application. But they require low-level process tracing capabilities that are highly O/S dependent and not widely available. It is also very difficult to implement robust system call interposition and it is unable to support non-POSIX compliant operations (e.g., setuid).

Compared to the above approaches, DFS-based techniques utilized by GVFS are key to supporting a wide range of applications, especially the ones that *must be deployed without modifications to source code, libraries, or binaries*. Examples include commercial, interactive scientific and engineering tools and virtual machine monitors that operate on large, sparse data sets [17][18][19][20].

2.2 Traditional Distributed File Systems

Distributed file systems (DFSs), such as NFS [21][22][23] and CIFS [24], have been successfully deployed on local-area systems (e.g., computer clusters), for decades, enabling applications to transparently access large amounts of remotely stored data. Their architecture is typically designed in a client-server style. A *server* stores the data in its local disks and it runs the *DFS server* to provide the remote file service. The DFS server hides the actual implementation of its local data access (e.g., a local file system) and the actual location of the data, and presents a standard interface defined by the DFS protocol to service the remote file access from clients. A *client* is where the user or application that needs the access of remotely stored data is at, and it runs the *DFS client* to handle the remote data access for applications. The DFS client, together with other components of the O/S, offers a generic file system interface, which is the same for both local and remote data access, to applications and thus hides the complexity of bringing in data from a remote server.

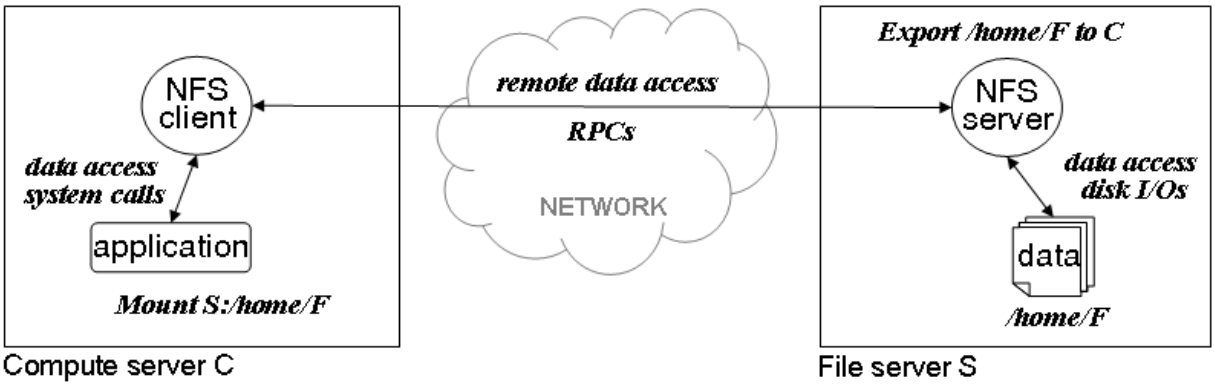

Figure 2-1. An NFS setup is typically created by mounting a file system from the server to the client, and the NFS-mounted file system is presented to applications in the same way as local file systems. An application's data access triggers system calls which are handled by the kernel, and they are passed on to the NFS client if the requested data are actually mounted from the remote server. The NFS client handles the data access by sending remote procedure calls (RPCs), according to the NFS protocol, to the NFS server across the network via either UDP or TCP. The NFS server processes the incoming client RPC requests and invokes I/Os on the server local disks to satisfy the access. The results are sent back from the NFS server to the NFS client and then returned to the application.

Network File System (NFS) is the *de facto* DFS since it is the most widely-deployed one. It has been implemented for a large number of different types of O/Ss, including UNIX, Linux, and Windows. The widely-used versions of NFS are NFSv2 and NFSv3, while the latest version, NFSv4, is also becoming available in the recent O/S distributions. The NFS clients and servers are typically implemented in O/S kernels. In NFS, access to remotely stored files is serviced in a *block-by-block* manner, that is, only the data blocks that are needed by a user or application are transferred across the network.

As illustrated in Figure 2-1, an NFS setup is typically created by mounting a file system from the server to the client, and the NFS-mounted file system is presented to applications in the same way as local file systems. An application's data access triggers system calls which are handled by the kernel, and they are passed on to the NFS client if the requested data are actually mounted from the remote server. The NFS client handles the data access by sending remote procedure calls (RPCs), according to the NFS protocol,

to the NFS server across the network via either UDP or TCP. The NFS server processes the incoming client RPC requests and invokes the necessary I/Os on the server's local disks to satisfy the access. The results are sent back from the NFS server to the NFS client and then returned to the application. This whole process is transparent to the application in that it is completely unaware of where the data are stored and how they are accessed, except for a probably longer delay of the access.

Many DFSs follow largely the same data access model as NFS. The DFS that is widely available on Windows-family O/Ss is the Common Internet File System (CIFS). It is based on the Server Message Block (SMB) protocol in which remote data access is carried out via SMB requests and responses between the CIFS client and server. Another noteworthy family of DFSs is AFS (Andrew File System [25]) and its descendants OpenAFS [26] and Coda [27]. Coda and earlier versions of AFS use a different data access model in which files are transferred entirely when they are accessed by applications. AFS and Coda are also designed with wide-area environments in mind and have special enhancements for such usage. The next section will discuss several important aspects of wide-area file systems in details, including cache and consistency, security, and reliability.

Similar to the GVFS approach proposed in this dissertation, several related systems have also leveraged user-level techniques based on loop-back server/client proxies to extend O/S-level DFS functionality — in essence, virtualizing DFSs by means of intercepting RPC calls of protocols such as NFS [21], e.g., the automounter [28], CFS [29], and SFS [30]. In particular, LegionFS [13] interposes a user-level modified NFS server between a kernel NFS client and a Legion server to provide access to grid objects. NeST [14] is a storage appliance that services requests for data transfers supporting a variety of protocols, including NFS and GridFTP. However, only a restricted subset of NFS operations and anonymous user access are available. Furthermore, the system does not integrate with unmodified kernel NFS clients, a key requirement for application transparency. The approach described in this dissertation differentiates from these efforts

in that it supports application-tailored DFSs which are important for data provisioning in grid-style environments.

2.3 Application-Tailored Grid File Systems

2.3.1 Need for Application-Tailored Enhancements

Transparency is the main motivation for using DFS-based techniques for grid-wide data access. However, currently there are no mechanisms that allow a conventional DFS implementation to be customized to support application- or user-tailored enhancements. To illustrate with examples, consider the case of a file server exporting user home directories to clients. Suppose user Alice is a programmer that uses a single client to perform the bulk of her software development (editing, compiling, debugging). User Bob is a researcher that uses one or more clients to develop signal-processing algorithms that later are to be run across many clients concurrently, using as inputs a large number of benchmark media files.

Existing DFSs are unable to recognize per-session and per-application differences that could drive performance and functionality improvements to these users. Consider the case of NFS. Currently widely deployed versions of the protocol (v2, v3) do not store client state information in the server, rely on client-initiated revalidation requests to check for consistency, and write-through cache blocks on file closes.

In Alice's example, an NFS client would not be able to exploit the fact that she uses a single client to aggressively cache read/write data in local disk, and thus could not avoid the unnecessary network calls for consistency checks and write requests. Neither would NFS clients be able to exploit the fact that Bob's input files do not often change and would poll the server to revalidate each individual file upon opening. A currently available customization — increasing cached attribute expiration times — would not be advisable because it would apply to the entire remotely-mounted file system, potentially forcing large expiration times on files of other applications/users that share the same file system.

These examples highlight cases where DFS behavior can be tuned by exploiting application knowledge such as:

Number of clients. For instance, aggressive caching of attributes and data can be performed without consistency checks if it is known that only a single client is associated with a particular computing session.

Sharing role of clients. For instance, consistency models well-suited for scenarios with one or a few writer clients and many reader clients can be performed if this property is known to hold true by middleware for a particular computing session.

Other examples of application knowledge that is important to DFSs include the application's need on the use of full-file or partial-file access, the strength of security enforcement, and the level of fault tolerance. The inability of performing optimizations based on such information presents a hurdle to the deployment of pervasive LAN file systems (e.g., NFS) across grid-style environments. As illustrated in the above Alice's and Bob's examples, if DFSs are capable of leveraging application knowledge, the number of client-server interactions can be reduced, thereby reducing server loads and average request latencies. However, typical DFS implementations are not designed to exploit such knowledge, for two important reasons.

First, traditionally DFSs are setup by system administrators, who, for management efficiency reasons, favor static, long-lived, homogeneous configurations at the granularity of a collection of users rather than dynamic, short-lived, customized setups at the granularity of an application session. In addition, current systems have no mechanisms allowing users to convey information about DFS features that they desire for their applications to system administrators.

Second, integrating application-tailored features with DFS implementations in commonly available kernels is very difficult in practice. For designers of a stable kernel tree, selecting which enhancements should be added based on application needs is difficult: an optimization tailored for one application (e.g., aggressive pre-fetching of

file contents) may result in performance degradation for several others (e.g., sparse files, databases). Furthermore, it is difficult for the kernel to gain application knowledge that is needed for driving the usage of such features — it may require additional system calls or non-standard APIs that must then be present in future releases for legacy support, even if the features are rarely used. In addition, kernel-level modifications (even if encapsulated into modules) are difficult to port and deploy, notably in shared environments. For management and security reasons, administrators are often strict about controlling their kernel configurations and are reluctant to allow modifications that deviate from stock O/S distributions.

The proposed GVFS-based approach addresses the need for application-tailored data provisioning by enabling user-level per-application customization on remote data access. The lack of support for application-tailored optimizations has also been recognized as a limitation by BAD-FS [10], which exposes the control decisions on caching, consistency, and replication to grid middleware. However, it relies on system-call based interposition agents, and therefore, as discussed in Section 2.1, it only supports specific types applications and O/Ss. In contrast, the techniques described in this dissertation enable application and O/S transparent grid file systems with application-tailored enhancements. These enhancements cover several important aspects of remote data access, including caching, consistency, security, and fault tolerance, which are discussed in details in the rest of this section.

2.3.2 Caching and Consistency

Caching is a classic, successful technique to improve the performance of various types of computer systems by exploiting temporal and spatial locality of data references and providing high-bandwidth, low-latency access to cached data. The basic idea of (client-side) caching is that when a client requests data from a storage facility, it temporarily stores a copy of the data in another layer of storage, namely cache, which is closer to the client and provides faster data access than the original storage.

Different levels of caching exist in a typical computer system. For example, CPUs use hardware-implemented caches to speed up the access to data stored in memory; O/Ss manage part of memory as caches to improve the access times to data stored on disks; and a DFS can use disks as caches to provide high-performance access of data from the remote file server.

Caches leverage the locality that typically exists in data references to improve its performance. There are two types of locality: temporal locality refers to that if a client accesses some data, it is highly probable that these data will be used again by the client in the near future; spatial locality means that when a piece of data is accessed, its spatially nearby stored data are also very likely to be needed by the client. If a data request can be satisfied from the cache, it is called a *cache hit*; otherwise, it is a *cache miss*, and the requested data need to be satisfied from the remote storage. Apparently, a cache is effective when most of the data accesses can be served from the cache, i.e., the hit rate is high and miss rate is low. A cache is initially "cold", which means it does not have any data to serve any requests; and as data are brought into the cache, it becomes "warm" and able to satisfy data requests leveraging the locality.

Different caching policies are possible: *read-only caching* only stores the data requested by read operations in caches, whereas *write caching* also caches the data accessed by write operations. There are two types of write caching: *write-through caching* allows a client to directly modify data in its cache and forward the update to the remote storage at the same time; *write-back caching* further delays the propagation of data updates and keeps the modified data only in caches for a period of time. Read-only caching works well for read-mostly data accesses, and it is also easy to be implemented in a robust manner. Write-through caching potentially offers improved performance over read-only caching as the data cached from writes can be reused by the following reads. Write-back caching is important to the data access performance when there are intensive writes, because the locality existed across write operations can be leveraged to reduce the

amount of data updates on the remote storage. It is, however, more complex to implement and more difficult to handle client failures and maintain data consistency between the cache and remote storage.

Cache consistency concerns that when there are multiple clients sharing the data stored on the remote storage, a client's read on a piece of data should always return value from the latest write on it, no matter whether the write is from the same client or others. Inconsistency happens when a client reads data from its cache while the data are already modified by another client, and when a client delays its updates in its cache while the other clients get a stale copy of the data from either their caches or the remote storage. Due to the lack of an absolute global time, it is impossible to determine which operation is the "latest" one, so a formal definition of cache consistency typically considers that the operations from all the clients on the same piece of data follow a hypothetical serial order. In this serial order, operations from any particular client follow the order in which they are issued by the client, and the value returned by each read operation is the value written by the last write to the piece of data in the serial order. This consistency model is able to present a coherent view of data to clients, but it can be very expensive to implement in practice. Existing DFSs often use other cache consistency models which are more or less relaxed from this one and provide relatively weaker data coherence.

Conventional DFSs usually employ client-side caching in memory, but the use of disk caching is not typical, since most of them are designed for local-area environment where the latency of network transactions is comparable to the latency of local disk access. For example, it is common among different NFS client implementations to cache file data blocks, attributes, file handles, and directories in memory, but disk caching is only available on Solaris with the kernel-level CacheFS [31] service. There are several related kernel-level DFS solutions that are specially designed for the use in WAN and exploit the advantages of disk caching. In particular, AFS [25][26] and Coda [27] make use of disk caching to improve both performance and availability. Coda and the earlier versions of

AFS cache files entirely on local disks, whereas the later versions of AFS also support partial-file caching (i.e., cache only certain data blocks of files on demand).

As caching is widely used in DFSs, cache consistency is an important task for DFSs to support the concurrent sharing of data for distributed clients. Traditionally NFS replies on a timestamp-based algorithm to maintain consistency of cached data. When a client caches any block of file (in the data cache), it also stores the file's modification time (in the attribute cache). The cached blocks of the file are assumed to be valid for a finite interval of time, and the first reference to any block of the file after this interval forces a revalidation, in which the client compares the recorded timestamp with the file's modification time on the server. If the later is more recent, it means that the file has been recently modified by someone else, so the client invalidates the cached blocks of the file and refetches them on demand. Because of this timestamp-based algorithm, a client needs to periodically revalidate a file if it is continuously referenced.

The NFS protocol also provides a close-to-open consistency in which a client always revalidates a file when it opens it and always flushes the locally modified data of a file when it closes it. This consistency model is useful for the "*sequential write-sharing*" scenario, in which a shared file is never open simultaneously for reading and writing by different clients. It makes sure that a client always gets the latest copy of a file when the client starts to work on it. However, if a file is open simultaneously by several clients and one of them modifies it, which is called "*concurrent write-sharing*", a stronger consistency model is needed to allow the other clients to see the changes immediately.

Several solutions are proposed to improve upon NFS and provide stronger cache consistency. Spritely NFS [32] applies the cache consistency protocol designed in Sprite [33] to NFS: it adds open and close calls to the NFS protocol to allow a server to keep track of the clients that open the file for reading and writing; and when the write-sharing of a file is detected, the server uses callback calls to inform the clients that the file is no

longer cacheable and force them to invalidate and/or write back their cached data of this file.

The NQNFS [34] uses leases to allow a client to cache data for reading or writing without worrying about conflicts. Such a lease has a limited duration and must be renewed by the client if it wishes to continue to cache the data. A read-caching lease allows a client to use read-only caching; a write-caching lease permits write-back caching, and the cached modifications are submitted when the lease expires or is terminated by an eviction callback (issued from the server when the sharing conflict is detected).

The most recent version of NFS (v4 [35]) differs from the earlier versions by including open and close calls in the protocol and provides open delegations to clients. With a read delegation, the client can use cached data without periodic consistency checks; a write delegation further allows the client to retain modified data in its caches. A lease is associated with every delegation and its expiration automatically revokes the delegation. When a sharing conflict is detected by the server, it can also revoke the delegation using a server-to-client callback call.

The cache consistency model provided by AFS [25][26] and Coda [27] is similar to the aforementioned close-to-open model, in which a client gets the latest copy of a file on open and propagates the modified file on close. There are two limitations of this model: first, modifications on a file cannot be retained in cache after the file is closed; second, it cannot support concurrent file sharing, since the modification made by a client cannot be seen by the others immediately.

In AFS and Coda, consistency is maintained by means of callbacks. When a client caches a file, the server keeps track of that and promises to inform the client if the file is modified by other clients. With this promise, the client can use the cached copy without checking the server. When a client updates the file, the server sends out callback break messages to the other clients, which have cached this file, since the server will discard the callback promises it held for these clients on this file. On the other hand, if a client opens

a file and finds it in its cache, it needs to check with the server whether the promise still holds. If not, it has to to fetch the latest version of the file from the server.

However, the above caching and consistency designs require kernel support that is difficult to deploy across shared grid environments, and they are not able to employ per-user or per-application cache policies. In contrast, GVFS enhances caching and consistency at user-level based on the virtualization of widely available NFS versions (v2 and v3), and supports per-user and per-application customization on the use of disk caches and consistency protocols.

Caching or replication on persistent storage is also widely used among the related scalable distributed data storage/delivery systems. Pangaea [36] is a decentralized wide-area file system that uses pervasive replication in a peer-to-peer fashion to improve system performance, but it supports only one consistency model — "eventual consistency", i.e., it only promises that a user sees a change made by another user in some unspecified future time. OceanStore [37] is an architecture designed for global-scale persistent storage; Pond [38] is its prototype that implements a file system interface using NFS loop-back server and allows for application-specific consistency, but it is not application-transparent, requiring the use of its API to achieve this goal. In the context of Web content caching, a related proxy cache invalidation approach has been studied in [39]. These systems differ from the proposed GVFS approach in that they are not architected to provide different consistency models transparently to applications according to their requirements and usage scenarios.

Note that the consistency models and protocols discussed in this dissertation only consider the order of read and write operations on a single data item (e.g., a file) and do not consider the order of operations on different data items (e.g., all the files in the file system). In the terminology typically used in shared memory multiprocessor systems, a consistency model specifies the constraints on the order of operations on the entire data set, whereas a coherence model only considers that with respect to a single data item

[40]. However, this definition of cache coherence has been traditionally referred as cache consistency in the literature of DFSs and it is thus followed in this dissertation.

2.3.3 Security

Security for a DFS typically concerns both confidentiality and integrity: *confidentiality* refers to that data accessed through the DFS are disclosed only to authorized clients; *integrity* means that alterations to the data can only be made in an authorized way. There are several important mechanisms to protect the security of a DFS. *Encryption* makes use of cryptography to transfer data into something that an attacker cannot understand, and the encrypted data can only be decrypted by someone with the proper key. *Authentication* is used to verify the claimed identity of a party and it is typically also based on cryptography. After a party is authenticated, *authorization* is the process to check whether that party is authorized to perform the requested access on the data, and the *access control* can be performed by checking the party's identity against an Access Control List (ACL) which lists the permitted operations. *Integrity check* makes sure that the data are not altered by unauthorized parties, and it can be done using Message Authentication Code (MAC) or digital signatures.

Full-featured security needs to support all of the above mentioned security mechanisms: authentication, encryption, integrity check, and access control. Strong security in DFSs is often based two types of security systems: Kerberos [41] and Public-key Infrastructure (PKI [42]). A Kerberos system is built around Key Distribution Centers (KDCs). Users are organized into realms and each realm's KDCs are managed by its administrators. A user authenticates into her realm through the realm's KDCs, from which she obtains the tickets for secure access to the resources in the realm. Across-realm access requires the cooperation of the administrators in each realm to develop trust relationships and exchange per-realm keys. In PKI-based security, a public-key based certificate (e.g., X.509 [43]) along with its associated private key uniquely identifies a user. Two parties can use their certificates to establish mutual authentication and then create a secure channel for

access. Validation of a certificate is done by checking the signature of its issuer, and a trusted third party known as Certificate Authority (CA) is typically leveraged to issue certificates.

2.3.3.1 Security in distributed file systems

Existing DFSs have diverse security designs and strengths. Earlier versions of NFS (v2 [22] and v3 [23]) rely on UNIX-style authentication, using user and group IDs. Although stronger authentication flavors are defined in the specifications, they have never prevailed in deployments. There is also no support for privacy and integrity in these versions, and NFS RPC messages can be easily spoofed, altered, and forged. Complete support of security has not been available until the latest version (NFSv4 [35]), which mandates the support of RPCSEC_GSS [44], a RPC-layer security protocol based on the Generic Security Services API (GSS-API [45]). It is required that a conforming NFSv4 implementation must implement RPCSEC_GSS with two security mechanisms, one based on Kerberos (V5 [41]) and the other based on PKI (LIPKEY, Low Infrastructure Public Key [46]).

All NFS versions use an exports file to specify the hosts that are allowed to access an exported directory. The ACCESS procedure call was introduced in NFSv3 to provide fine-grained access control using POSIX-model ACLs, but again it is not widely used in practice. NFSv4 improves upon this by providing Windows NT-model ACLs which have richer semantics and wider deployments. In addition, NFSv4 represents users and groups with string IDs instead of integers, which facilitates cross-domain identity mapping.

The AFS [25][26] and Coda [27] use Kerberos-based systems to provide strong security. Access control is achieved by associating an ACL with directories that list positive or negative rights for a user or group. The Kerberos security relies on centralized control and works well within an intranet. But cross-domain security is difficult to set up because it requires the involved administrations to negotiate a trust relationship.

None of these conventional DFSs has been designed to support grid security requirements. There is related work on extending DFS security at kernel-level. In particular, the GridNFS [47] project develops a GSI-compatible security in NFSv4. However, such a design requires kernel support that is difficult to deploy across shared grid environments. Kernel-level security techniques are also unable to employ per-user or per-application security configurations. In contrast, a GVFS-style user-level solution can support flexible customization on security mechanisms and policies based on individual application and user needs.

User-level techniques can achieve privacy and integrity of NFS through secure tunneling, where SSH or SSL can be leveraged to establish a secure end-to-end connection between the client and server for NFS traffic [48]. A secure tunnel multiplexed by users faces the same limitations as NFS, since RPC-layer mechanism is still required for authentication and authorization within the tunnel, and such tunnels are created statically by system administrators. In GVFS, per-session SSH channels are created on demand to ensure privacy and integrity of the data sessions, whereas authentication and authorization can be performed by proxies using middleware-managed session keys.

Self-certifying File System (SFS [49]) also leverages user-level loop-back client and server to enhance DFS security. It addresses the problem of mutual authentication between a file server and users by providing self-certifying pathnames for files. Such a pathname has the server's public key embedded inside, which is used by a client to verify the authenticity of the server, and then to create a secure channel to protect the file system traffic. The SFS approach is also extended to provide decentralized access control, in which users are allowed to create file sharing groups with ACLs in the file system [50]. When a user tries to access a file, the authentication server fetches the user's credentials and check them against the ACL to authorize the access. Compared to SFS, the proposed GVFS focuses on providing data access that meets grid security requirements, employs dynamically-created per-user, per-application file system proxies,

and allows for middleware-controlled security configurations on a per-user, per-application basis.

2.3.3.2 Security in grid systems

The dynamic and multi-institutional nature of grid-style environments introduce new challenges to security. In [51] several key requirements were studied for a grid security model, including the support for multiple security mechanisms, dynamic creation of services, and dynamic establishment of trust domains. This research resulted in a *de facto* grid security standard, GSI (Grid Security Infrastructure). GSI employs public-key based certificates for grid authentication. Authorization is done by checking a grid user's identity (the distinguished name in the user's certificate) against certain access control mechanism (e.g., gridmap file in GSI, MayI layer in Legion [52]). One important security requirement unique to grid systems is delegation, which allows a service to act on behalf of a user. This can also be supported with extensions to public key certificates, e.g., proxy certificates in GSI and credentials in Legion.

Grid security can be implemented at two different levels. Transport-level security [53][54] uses public-key certificates to create a secure socket layer connection between two end points and protect the data exchanges between them. It is a mature technology that has efficient implementations (e.g., OpenSSL [55]), but it lacks service-level semantics and does not work for multi-hop connections. Message-level security is a suite of standards arising from the emerging Web service technologies [56][57][58], which provides security at the layer of SOAP messaging. It is agnostic to transport-layer protocols and connections, and supports more service-level functionalities. However, its performance is not comparable to transport-level security because XML processing is expensive. In this dissertation, a two-level security architecture that exploits the advantages of both levels of security is proposed for the GVFS-based grid data management.

In the related data management solutions, GSI-based GridFTP [6] provides API for programming grid data access, and RFT is a web service for reliable file transfer using

GridFTP; the Legion system [59] is an object-based grid system, which employs a modified NFS server to provide access to grid objects, and it integrates GSI in Legion-G [60]; the Condor system [9] uses library call interception to provide remote I/O, and it also supports GSI in Condor-G [61]. This dissertation proposes a grid-wide file system with compatible security mechanisms with these efforts. It differentiates from and complements them in that GVFS-based data sessions allow unmodified application binaries to access grid data using existing kernel clients and servers, and support application-tailored per-session customization.

2.3.4 Fault Tolerance

Reliable remote data access requires DFSs to tolerate the possible failures happened in the systems. This is especially important for grid/wide-area file systems because of the dynamic nature of such environments. The common types of failures include server and client crashes due to software or hardware problems, as well as network partitioning caused by crashed network devices which break the physical network connection between the client and server. Another type of failures is data corruption happened during data storage and transmission, causing incorrect results from data access. In addition, resources can also become unavailable to a DFS when resources voluntarily leave the system, which is common in grid and peer-to-peer systems built upon non-dedicated resources. In the worst possible failure semantics, any of the above types of failures may occur and a client cannot tell whether the result received from a server is correct or not — such a scenario is referred to as arbitrary or Byzantine failure.

Fault-tolerant systems are often built by replicating the data to introduce redundancy into the system. Server failures can be masked if the data are replicated across different servers, and tolerance of network partitions can also be provided by replicating the data across different sites. Successful recovery of an application's execution after a client failure often relies on the use of checkpointing mechanism, which saves the state of the application on persistent storage. After the client comes back from a failure, the

application can roll back to its most recent checkpoint and continue its execution from there.

There are two basic models of replication, passive (primary-backup) and active replication. In the *primary-backup replication* model, only the primary replica services data requests and it synchronizes with the backups by sending the updated data to them. If the primary replica fails, one of the backups is promoted to act as the primary. In *active replication*, all replicas execute operations in the same order, which usually causes higher overhead compared to passive replication, but it can tolerate Byzantine failures by collecting the results received from the replicas and using voting to find out the correct one.

Conventional DFSs have limited support for fault tolerance. AFS [25][26] supports read-only replication of data that are frequently read but rarely modified, in order to enhance data availability; Coda [27] supports read-write replication with a read-one, write-all approach. Earlier versions of NFS (v2 [22] and v3 [23]) do not provide any support for replication; with the help from Automounter [28], a remote mount point can be specified as a set of servers instead of a single one which allows the use of replication, but propagation of modifications to replicas has to be done manually. FT-NFS [62] is a user-level NFS that employs a primary-backup replication scheme to improve data availability. BFS [63] is another NFS service which employs a replication algorithm for tolerating Byzantine faults. The latest version of NFSv4 [35] provides very limited support for using read-only replication: each file can have a attribute to list the file system locations where the file's replicas are stored, but the management of replicas is left out of the protocol.

Fault-tolerance techniques are widely used in large-scale distributed storage systems. Oceanstore [37] and its prototype Pond [38] encode data with an erasure code to introduce redundancy and spread the coded data over a large number of servers to provide high availability. The PAST project [64] is peer-to-peer storage system that uses replication for

durability. The FarSite system [65] aims to build a scalable serverless network file system, using replication to provide file availability and reliability. Wide-area data replication is presented in [66] for scientific collaborations. It manages grid data replication for read-only scientific data sets using Globus Reliable File Transfer service for scheduling of GridFTP-based data transfer, and using Globus Replica Location Service for locating data replicas.

Compared to these systems, the GVFS-based approach described in this dissertation supports application-tailored customization on fault-tolerance mechanisms and policies, and it leverages middleware services for autonomic replication management and optimization.

2.4 Service-Oriented and Autonomic Data Management

Service-oriented architecture (SOA) is an approach to building loosely coupled distributed systems with minimal shared understanding among system components. In particular, the Web services architecture [67] has been broadly accepted as a means of structuring interactions among distributed software services, which exchange XML documents using SOAP messages over a network. Web Service Resource Framework (WSRF [68]) is a specification that describes a consistent and interoperable way of dealing with stateful resources that typically exist in a grid system, e.g., files in a file system and records in a database. This framework is becoming widely adopted by grid middleware including Globus Toolkit version 4 [69], WSRF.Net [70], and WSRF::Lite [71]. The system described in this dissertation focuses on data management and is unique in the support for dynamic and customizable data sessions, and it can also provide interoperable service to other grid middleware services based on WSRF.

Autonomic computing addresses the complexity of managing large-scale, heterogeneous computing systems by endowing systems and their components with the capability of self-managing according to high-level objectives [72]. The building blocks of an autonomic system are autonomic elements. An autonomic element manages its own

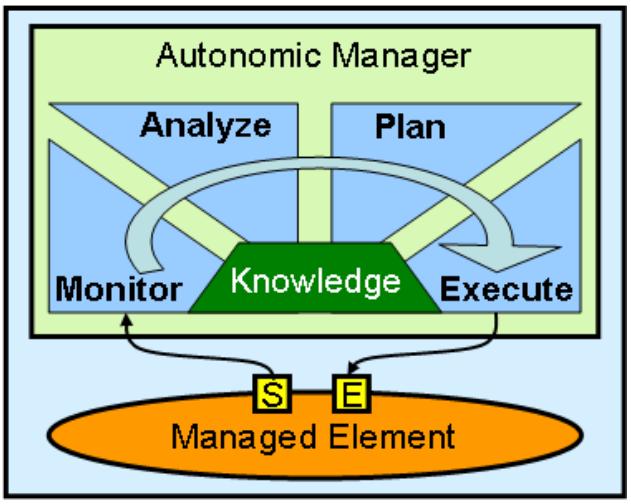

Figure 2-2. An autonomic element employs an autonomic manager to monitor the managed element, analyze the monitored information, plan management actions upon the element, and execute the plan accordingly. Self-management of the element is realized through this feedback-control loop.

resource or service guided by policies, and its typical architecture is illustrated in Figure 2-2. It employs an autonomic manager to monitor the managed element, analyze the monitored information, plan management actions upon the element, and execute the plan accordingly — self-management of the element is realized through this feedback-control loop. Furthermore, such autonomic elements also interact with each other to achieve the desired system-level self-management [73]. The proposed research follows this approach by building grid data management services as self-managing interacting autonomic elements.

Automatic performance optimization and fault tolerance are proposed in [74] for file staging based data provisioning. In this approach, performance improvement is through the tuning of data block size and TCP parameters; failure recovery is achieved by logging transfer progress and retry it after a failure. In comparison, the approach of this dissertation supports general data access patterns beyond bulk data transfer, and in particular, it can support interactive applications and efficient sparse file accesses. It also supports flexible customization and autonomic optimization on a variety of important aspects of data sessions based on application needs.

There is extensive research on autonomic storage management. In particular, in [75] a utility-based algorithm is used to decide the replication degree (the number of replicas for a data set) for resource managers; the IBM autonomic storage manager implements policy-based storage allocation [76]. Automatic replica generation and distribution are studied in the context of Content Delivery Network (CDN) [77] and peer-to-peer storage systems [78]. Compared to these systems, this dissertation proposes autonomic storage and replica management in order to support dynamic grid-wide file systems that provide application transparent and tailored grid data access.

2.5 Support for Distributed Virtual Machines

A virtual machine (VM) presents the view of a duplicate of the underlying physical machine to the software that runs within it, allowing multiple operating systems to run concurrently and multiplex the resources of a computer, including processor, memory, disk, and network. Such VMs are often called system-level VMs, in order to differentiate with other types of VMs, and they are becoming increasingly valuable to provide flexible resource containers and portable encapsulations of execution environments. In particular, there are growing interests in employing VMs in grid computing [17][79].

System-level VMs are mainly provided by the software called virtual machine monitor, also known as hypervisor (sometimes also with certain level of hardware support). A VM's entire state, including CPUs, memory, and disks, can be represented as data. In typical VM technologies, such as VMware [80], Xen [81], and UML [82], a VM's state data are often encapsulated in files and stored on physical disks. Thus the GVFS-based approach proposed by this dissertation can be applied to manage VM state and provide remote state access for VMs instantiated across grids.

There is a related project which has investigated techniques that improve the performance of VM migrations [83][84]. Their work focuses on mechanisms to transfer the state of virtual desktops, possibly across low-bandwidth links. Common between their approach and this dissertation are mechanisms for on-demand block transfers, and

optimizations based on the observation that zero-filled blocks are common in suspended VM memory state. The key differences are: the techniques described in this dissertation are generic to various VM technologies, implemented through intercepting NFS RPCs and leveraging O/S clients and servers available in typical grid resources, whereas their approach is specific to a particular type of VM, using modified libraries as a means of intercepting VM monitor accesses to files with a customized protocol.

The work presented in [85] introduces techniques for low overhead migration of VM memory state in LAN environments. The approach presented in this dissertation provides an efficient way of migrating VMs across the WAN. In addition, the implementation of [85] requires access to the VM's shadow page table, which is not possible for commercial VM software, such as VMware. On the contrary, the techniques described in this dissertation are applicable to different types of VMs.

CHAPTER 3
DISTRIBUTED FILE SYSTEM VIRTUALIZATION

Conventional distributed file systems (DFSs) are designed for general-purpose usage, and implemented in operating systems (O/Ss) (as part of kernels or privileged user-space software). They are unable to incorporate application-tailored features, because an optimization tailored for one application (e.g., aggressive pre-fetching of file contents) may result in performance degradation for several others (e.g., sparse files, databases). Modifications to DFSs at O/S-level are also difficult to port and deploy, notably in shared environments. On the other hand, such DFSs are unable to employ configurations that are customized for specific applications: they are typically deployed by system administrators with relatively static, long-lived, and homogeneous configurations at the granularity of a collection of users, rather than dynamic, short-lived, and different setups at the granularity of an individual user or application session.

This dissertation addresses these limitations by proposing virtual DFSs, namely **G**rid **V**irtual **F**ile **S**ystems (GVFSs) (Figure 3-1). These virtual DFSs are built upon the conventional DFSs, but they can behave differently than the physical ones in terms of data accessibility and characteristics. They share the underlying software and hardware resources, but they are isolated from each other and can be dynamically created and configured independently. With GVFS, application-tailored data provisioning can be realized by establishing per-application virtual DFSs and customize each of them according to its application's requirements and characteristics.

3.1 User-Level Proxy-Based Virtualization

3.1.1 Architecture

A GVFS-based virtual DFS consists of several virtual clients and servers which are implemented by unprivileged user-level file system proxies. These proxies virtualize the physical O/S-level DFS clients and servers by interposing between them and broker the data sharing across the distributed systems (Figure 3-2): the O/S servers delegate the

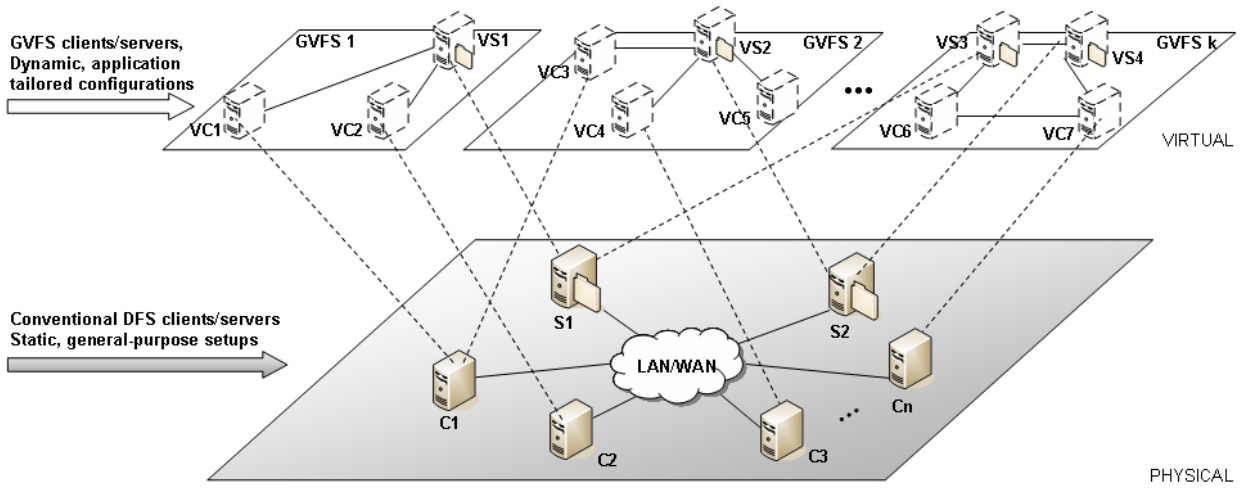

Figure 3-1. The GVFS-based virtual DFSs are built upon the conventional DFSs, but they can behave differently than the physical ones in terms of data accessibility and characteristics. They share the underlying software and hardware resources, but they are isolated from each other and can be dynamically created and configured independently.

control of data sharing to the proxies on the server-side (namely, *proxy servers*), and the O/S clients access the remote data through the proxies on the client-side (namely, *proxy clients*). The layer of indirection provided by the proxies forms the *virtualization*, which allows the proxies to multiplex the physical DFS clients and servers and to establish independent virtual DFSs for applications, with access to different data sets and with different configurations.

The GVFS can leverage the widely available DFS implementations in existing O/Ss, such as NFS and CIFS/Samba, to provide virtual DFSs across heterogeneous platforms. Network File System (NFS [22][23][35]) is the *de facto* DFS, available on many O/Ss, including UNIX and Linux as well as Windows (with an extension service). Common Internet File System (CIFS [24]) is provided by the Windows-family O/Ss and is interoperable with Samba [86] on UNIX and Linux. By virtualizing these DFSs with user-level proxies, GVFS can be seamlessly deployed on a wide variety of systems, without any changes to the existing O/Ss.

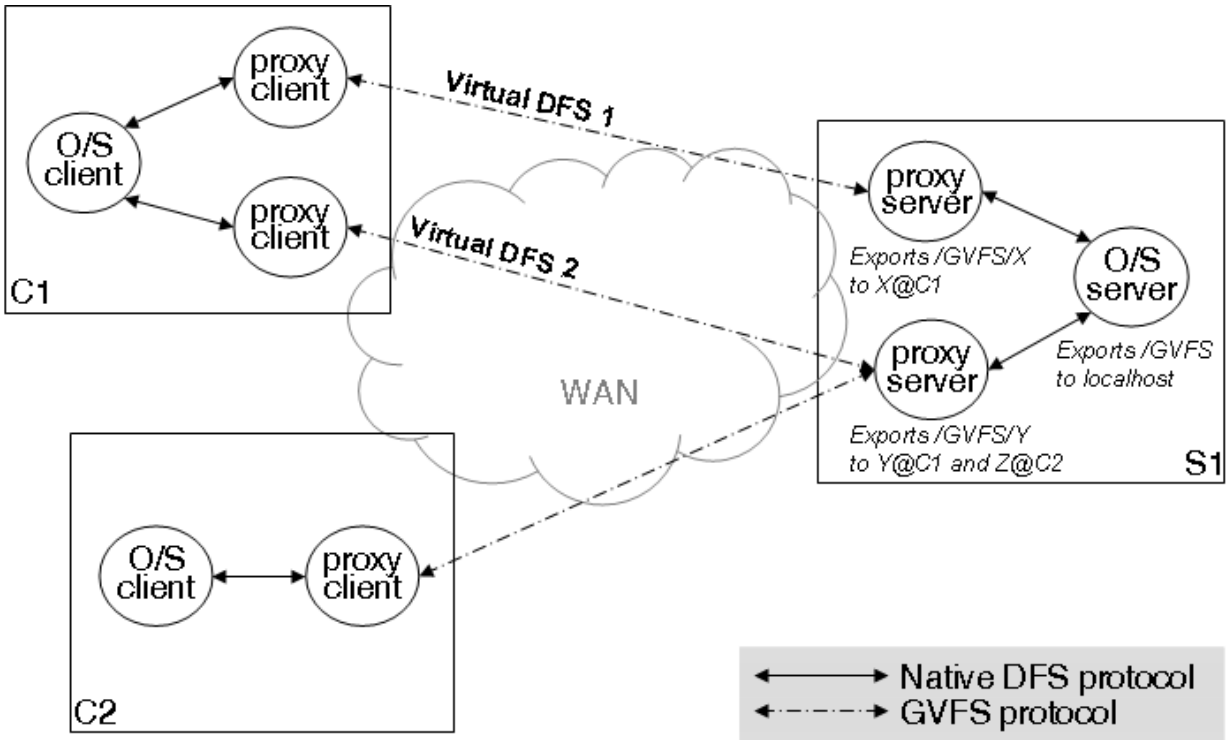

Figure 3-2. A GVFS-based virtual DFS consists of several virtual clients and servers which are implemented by unprivileged user-level GVFS proxies. These proxies virtualize the physical O/S DFS clients and servers by interposing between them and broker the data sharing across the distributed systems.

Unlike a conventional DFS statically deployed in a local-area environment, GVFS can be dynamically created across wide-area networks and administrative domains. In a wide-area environment, a user's identity is often not consistent across domains due to the lack of centralized administration. In a grid system, virtual organizations are dynamically established on resources distributed across different physical organizations, where a grid user's identity needs to be dynamically mapped to the physical ones. Therefore, an important task performed by a GVFS proxy is cross-domain identity mapping, which dynamically maps the identities between the account where the job is running and the account where the files are stored.

While virtualizing a conventional DFS, the GVFS proxies communicate with the O/S clients and servers via the native DFS protocols (e.g., NFS RPC, CIFS SMB). Nonetheless, the protocol used between the proxy clients and servers can be different

from the native protocol. It can be extended to provide more functionality and improved to achieve optimization on various important aspects of remote data access, including performance, consistency, security, and fault-tolerance.

Based on the GVFS approach, both NFS and CIFS can be virtualized by interposing file system proxies between native NFS and CIFS clients and servers. The resulting virtual NFS's or CIFS's data access is managed by its proxies, which process and forward the corresponding NFS RPC or CIFS SMB messages. Because of the good standardization and availability of the NFS specifications [22][23][35], this dissertation uses *NFS-based GVFS* to present its design and implementation, as well as to develop and evaluate the prototype system.

3.1.2 NFS-Based GVFS

3.1.2.1 User-level NFS proxy

In a typical NFS setup, the kernel NFS server exports the shared file systems to certain users and clients, and an authorized user access the remote data via the kernel NFS client's RPC. With GVFS, a virtual NFS can be created by placing a user-level file system proxy between the kernel NFS client and server. To the kernel client, the proxy works as a server; to the kernel server, the proxy works as a client. When a user or application on the client needs to access the remote data on the server, the RPCs from the kernel NFS client are processed by the proxy and then forwarded to the NFS kernel server.

Specifically, due to the decoupling of mount protocol and NFS protocol (in NFSv2 and NFSv3), a proxy consists of two user-level daemons, *gvfs.mountd* and *gvfs.nfsd*, for handling mount RPCs and NFS RPCs respectively. When a client tries to mount the remote file system, the request is processed by the GVFS mount daemon which first checks whether the client is allowed to mount it. If it is allowed, the proxy forwards the request to the server's native NFS mount daemon, and the result from the server (the root file handle of the remote file system) is returned to the client. Otherwise, the request is rejected by the GVFS mount daemon and is not forwarded to the server.

A GVFS is established once the remote file system is mounted on the client. A user's data access to the remote file system on the client triggers the kernel NFS client to issue RPCs to the GVFS proxy's NFS daemon. The proxy checks whether the client and user are allowed to access the requested data, forwards the permitted ones to the native NFS server, and returns the results to the client. Invalid requests are rejected by the proxy without being forwarded to the server. In the end, a GVFS can be destroyed by unmounting the remote file system and terminating the proxy daemons.

The GVFS enforces access control for the mount and NFS requests by checking the user's identity (typically user ID and group ID) and the client's identity (typically the IP address) against its access control list files. It uses an exports file to list the clients that are allowed to access the remote file system and their read and write permissions. Another map file is used by GVFS to list the users that are allowed to access the remote file system.

This map file also stores the cross-domain user identity mappings between the account where the user's job is running and the account where her files are stored. Upon receiving a request, the proxy checks the identity of the job account (embedded in the RPC message) against the map file and finds out the corresponding file account's identity. It then changes the user and group IDs of the job account to the IDs of the file account in the the RPC message before forwarding it to the server. If a job account's identity is not found in the map file, the request is either denied or the account is mapped to *nobody* (the least privilege account) depending the configuration of GVFS.

Figure 3-3 illustrates an example setup of two NFS-based GVFSs. Grid user X and Y are running their jobs under account *shadow1* and *shadow2* on computer server *C1* and *C2* respectively. On the file server S, the corresponding user data are located in subdirectory X and Y of file account F (*/home/F*). The native NFS server exports user data to to itself (*localhost*) since the proxies are running on the file server. When a job needs to access the remote data on GVFS, the corresponding requests from the kernel

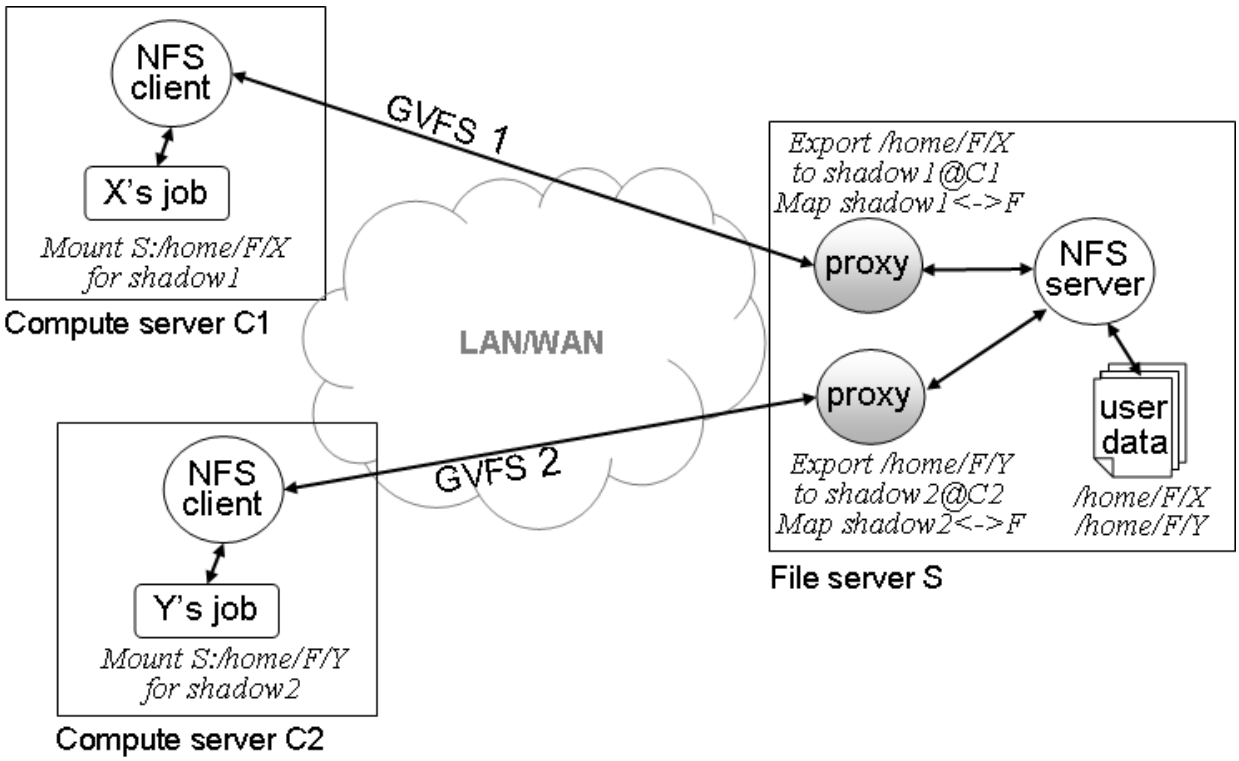

Figure 3-3. Two grid users (X, Y) execute jobs with their allocated shadow accounts (*shadow1*, *shadow2*) on computer server *C1* and *C2*, respectively. They access their data remotely stored on file server S under the file account F. Their data requests are authenticated and processed by the user-level GVFS proxies on S. The accepted requests are forwarded to the NFS server, and their credentials inside the requests are mapped from the shadow accounts to the file account.

NFS client are processed by its proxy. For a valid request, the proxy modifies the RPC message to map the job account's identity (*shadow1* or *shadow2*) to the file account's (F), so that the request can be properly serviced by the kernel NFS server. Upon receiving the result from the server, the proxy also modifies the RPC message to map the file account's identity back to the job account's and then forward it to the client. Because the user data are only exported to *localhost* on the server, the remote data access is completely under the control of the proxies: the clients do not have direct access to the data; they can only access them through the proxies. The path exported by a proxy also ensures that a user (e.g., X) can only access her own data (*/home/F/X*); she cannot access the other users' files (e.g., */home/F/Y*) since they are not exported to her client by the proxy.

3.1.2.2 Multi-proxy GVFS

Although a GVFS-based virtual DFS can be created by leveraging a single native NFS server-side proxy [87], the design supports connections of proxies "in series" between a native NFS client and server. In particular, a pair of proxies can be placed on the kernel client and server to provide both virtual client and server as described at the beginning of this chapter. In this setup, the kernel server still exports the file system to its local proxy server (the proxy running on the server side), the proxy server exports it to the proxy client (the proxy running on the client side), and the proxy client exports it to the kernel client. So the client can only access the remote file system through the proxy client: the proxy client forwards the access to the proxy server, and the proxy server then forwards it to the kernel server. The proxy server also performs necessary user identity mapping to support cross-domain data access, which is not needed for the proxy client.

As illustrated in Figure 3-4, a pair of proxy client and server cooperate between the native NFS server and client to establish a GVFS (e.g., *GVFS1*). The proxy server works in the same way as in a single-proxy GVFS: the kernel NFS server exports the user data directory (*/home/F/X*) to *localhost*, so the proxy server on S is responsible for processing and forwarding the remote access to the data (it also maps the job account's identity *shadow1* in the RPCs to the file account's identity F). On the other hand, the client accesses the remote data through the proxy client on C. The proxy client exports the remote file system to its *localhost*, so it accepts only the requests coming from the kernel NFS client and it then forwards them to the proxy server.

Although a multi-proxy design may introduce more overhead from processing and forwarding RPC calls, there are important design goals that lead to its consideration:

Improved performance: The addition of a proxy at the client side enables techniques which address the inefficiency of native NFS protocol and improve the performance of remote data access. For example, a proxy client can introduce a level of caches on local disks additional to kernel-level memory buffers and improve access

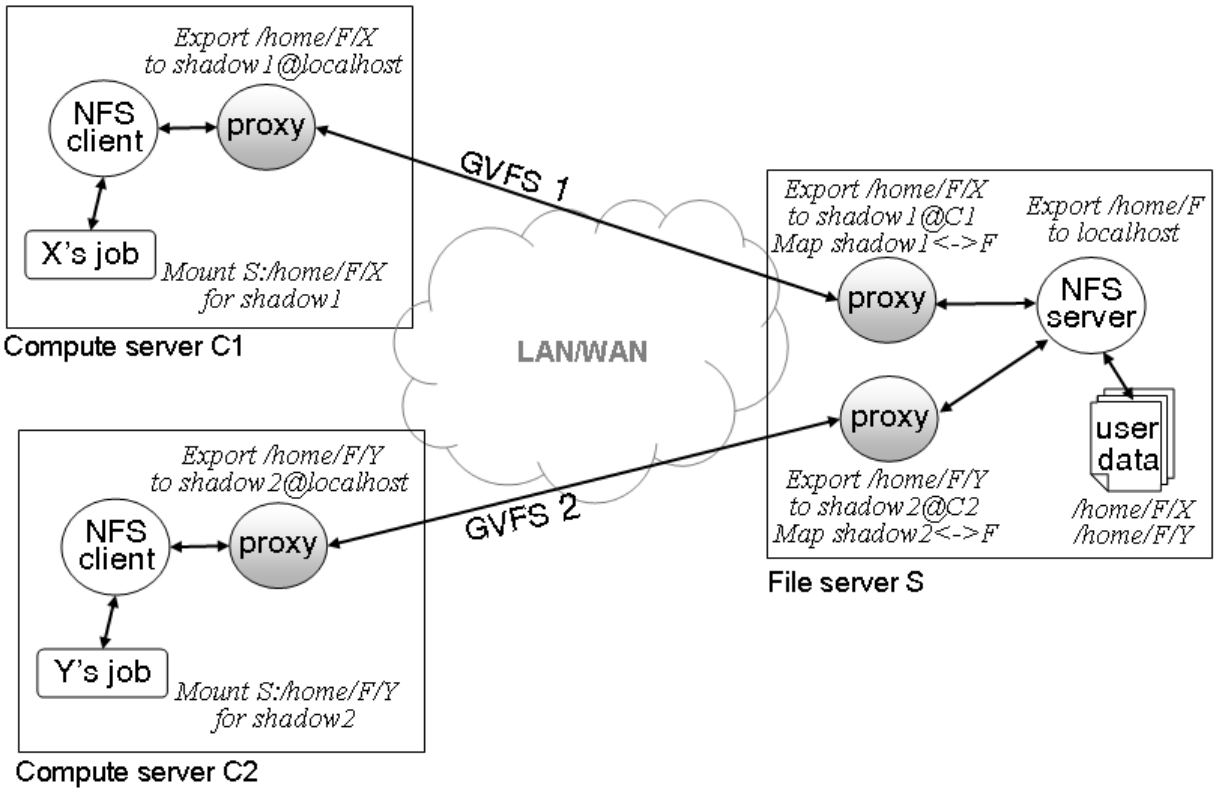

Figure 3-4. Multi-proxy GVFS setup. Two proxies work between the native NFS server and client cooperatively to provide remote data access. Kernel NFS server exports the user data directory (*/home/F/X*) to *localhost*, so the proxy server on *S* is responsible for processing and forwarding the remote access to the data. It also maps the job account's identity *shadow1* in the RPCs to the file account's identity *F*. On the other hand, the client accesses the remote data through the proxy client on *C*. The proxy client exports the remote file system to its *localhost*, so it accepts only the requests coming from the kernel NFS client and it then forwards them to the proxy server.

latency for requests that exhibit locality; the proxy client and server can also employ inter-proxy high-throughput data transfer protocols (e.g., Secure FTP, GridFTP [6]) for access of large files.

Additional functionality: Extensions to the NFS protocol can be implemented between proxies without modifications to native NFS clients/servers or applications. For example, secure remote data access can be achieved by inter-proxy authentication, authorization, data encryption, and integrity check; the proxy client and server can also

cooperate to realize fine-grained consistency models to maintain the data coherence between client-side caches and server.

These potential enhancements enabled by multi-proxy GVFS are discussed in detail in Chapter 4. The rest of this chapter presents a thorough performance evaluation of the basic NFS-based GVFS implementation.

3.2 Evaluation

3.2.1 Setup

A prototype of GVFS is implemented based on virtualizing NFS (v2 and v3) and it is evaluated in this subsection using experiments with typical file system benchmarks in a local-area environment. These benchmarks exercise GVFS with intensive file system operations and demonstrate its performance compared to conventional DFSs. The experiments were conducted on a high-speed LAN (Gigabit Ethernet) in order to reveal the worst-case overhead of the user-level virtualization. Two physical servers were used as the file system client and server, and each has dual 2.4GHz hyper-threaded Xeon processors with 4GB memory and runs Fedora Core 6 with kernel 2.6.17.

These experiments compare the performance of the NFS-based GVFS with the native NFS, where the version 3 of NFS over TCP was used for both. The servers exported the file system with write delay and synchronous access. The native NFS daemon used the default configuration of 8 threads, whereas the GVFS proxies were also multithreaded with 8 worker threads. (See 4.2.2 for a detailed discussion on GVFS multithreading.) Due to the limitation of the kernels, the maximum block size for read and write RPCs was set to 32KB. No swap was used on the physical machines during the experiments. Every run was started with cold kernel buffer by unmounting the file system.

3.2.2 Stat

The virtualization provided by GVFS involves overhead from processing RPCs with the user-level proxies. The first experiment studies this overhead by using a micro benchmark to measure the latency of a single RPC. This benchmark uses the stat system

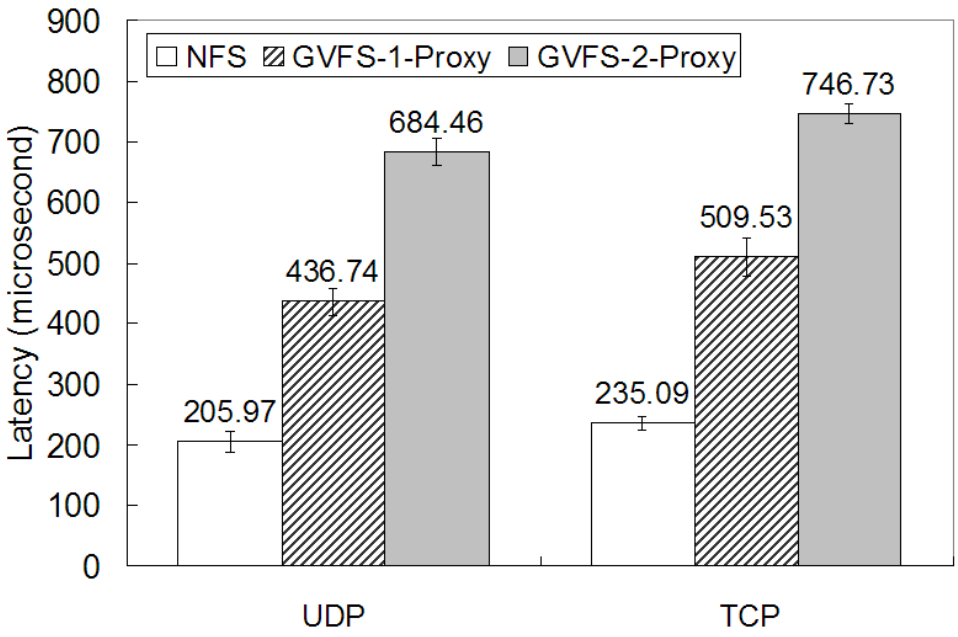

Figure 3-5. Latency of a stat system call that triggers a single GETATTR RPC to the file server. Three different DFS setups were considered: native NFS (*NFS*), GVFS based on only proxy server (*GVFS-1-Proxy*), and GVFS based on both proxy client and server (*GVFS-2-Proxy*).

call to check a directory on the remote file system, which triggers the kernel client to issue a single GETATTR RPC to retrieve the directory's attributes. These attributes are preloaded in the server's memory, and thus the latency of the stat call mainly entails the network round-trip time and the RPC processing delay.

Compared to using NFS to serve the stat call, GVFS introduces additional latency from the user-level RPC processing and the kernel-user space switching. To take a closer look at this, two different GVFS setups were tested: in *GVFS-1-Proxy*, only the proxy server was used to create the virtual DFS; and in *GVFS-2-Proxy*, both proxy client and server were employed. Both UDP and TCP were considered as the transport to carry the RPCs. Figure 3-5 illustrates the latencies of the stat call on these different setups. With a single proxy, GVFS adds 0.23 ms and 0.27 ms of delay with UDP and TCP, respectively. When both the proxy client and server are used, the virtualization overhead increases to 0.48 ms for UDP and 0.51 ms for TCP.

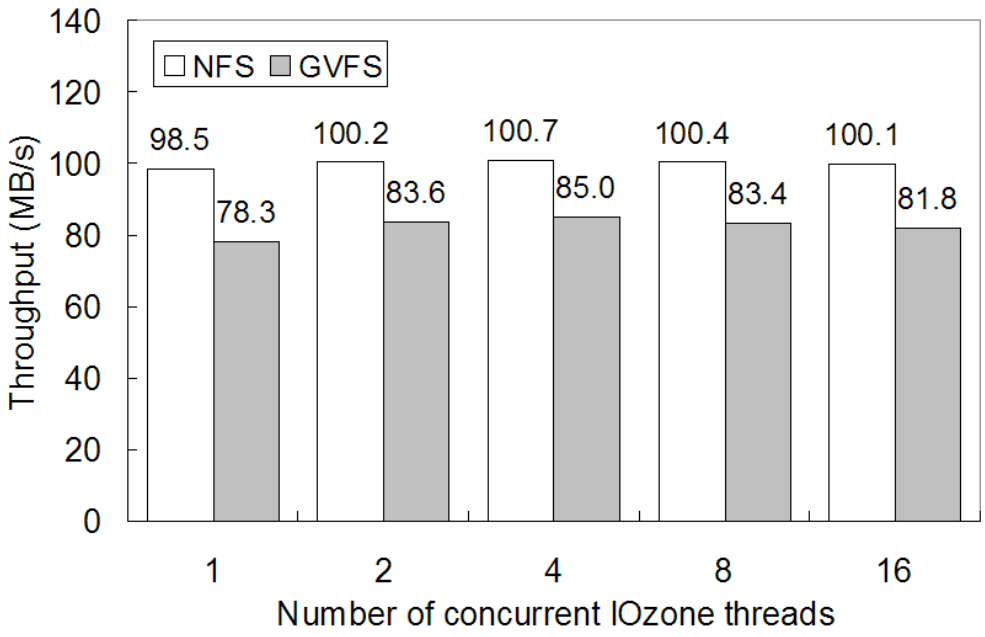

Figure 3-6. Throughputs of IOzone with different numbers of threads reading large files through separate NFS/GVFS connections to the file server. (The standard deviations are all under 1% of the reported average values.)

This micro benchmark shows that the latency for a single RPC is doubled or tripled using GVFS. However, the application perceived overhead can be much smaller, because the processing of multiple outstanding RPCs can be overlapped, and the latency of disk accesses can also diminish the user-level delay. This is demonstrated by the following experiments. The typical GVFS setup which utilizes both proxy client and server is used throughput the rest of this subsection.

3.2.3 IOzone

The second experiment evaluates the throughput of GVFS with a typical file system benchmark, IOzone [88]. It is used to sequentially read a large file (1GB) from the remote file system. The file is preloaded in the file server's memory, and thus there is no disk access to slow down the benchmark's request rate. This "extreme" intensive setup reveals the worst-case overhead from the user-level virtualization. The throughputs on NFS and GVFS are plotted in Figure 3-6 (the first group of bars). Note that the maximum TCP throughput between the client and server is 111MB/s (measured with Iperf [89]). In

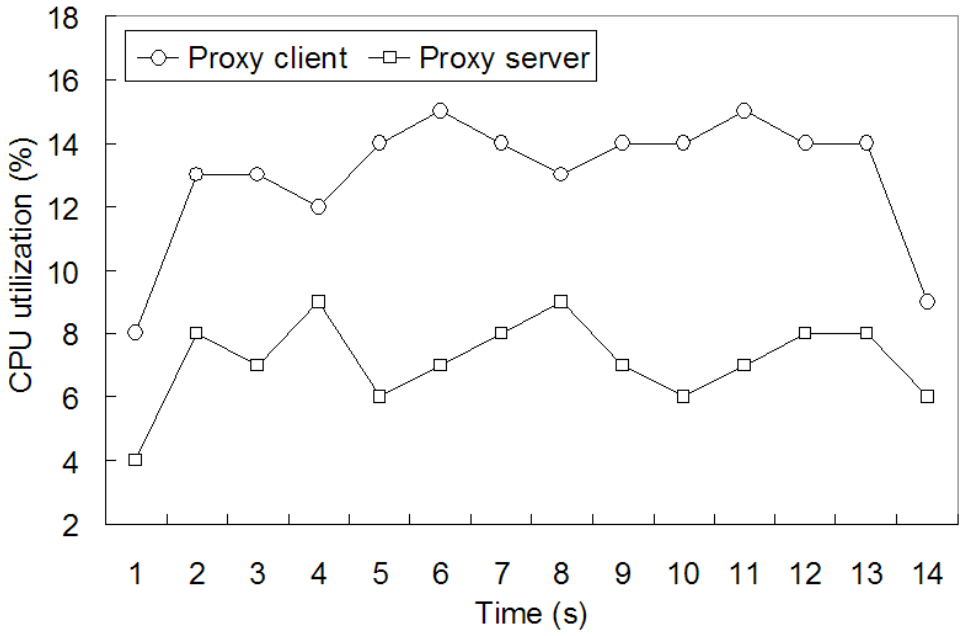

Figure 3-7. The CPU usage of the GVFS proxy client and proxy server during one typical run of IOzone on GVFS. The average user time percentages for the proxy client and server are about 14% and 8% respectively.

comparison, GVFS delivers a performance that is 70% of the maximum and 80% of NFS. This confirms that the capability of handling many outstanding requests helps GVFS to significantly reduce the overhead of user-level virtualization.

This overhead was further measured in terms of CPU usage of its user-level proxies. The user CPU time percentages for the proxy client and server were collected throughout the benchmark's executions. In average, they consume about 14% and 8% of CPU on the client and server, respectively. Figure 3-7 shows the proxy client's and proxy server's CPU consumptions for one typical run of the benchmark. Considering the intensity of the workload, these usages are reasonable, and they can be much lower for typical applications. The proxy client spends more cycles than the proxy server because its RPC takes longer to finish, which is serviced across the network, whereas the proxy server's RPC is replied from its localhost.

This experiment has also studied the scalability of GVFS with a multi-client setup, where a single proxy on the server services concurrent data access from multiple proxies on

the client. IOzone was executed with several threads, each sequentially reading a different file through a separate proxy client. As the number of threads was increased, the size of the files was reduced accordingly (from 1GB to 192MB), so they could still be preloaded in the server's memory. For comparison, the benchmark was also executed on native NFS, where separated connections between the kernel client and server were used for the IOzone threads to access files.

The aggregate throughputs with various numbers of threads are shown in Figure 3-6. Compared to the results from the previous single-client tests, the throughput of GVFS is consistent with respect to both the maximum achievable throughput and NFS' throughput. Regardless of the number of concurrent intensive clients, GVFS can always effectively utilize the resources and deliver the same level of performance. This demonstrates that GVFS can support scalable data sharing by using a single proxy server to service a large number of clients. It also shows that having several proxies running on the same host can be an efficient way of providing multiple virtual DFSs upon a single physical resource.

3.2.4 PostMark

The third experiment chooses a more realistic file system benchmark, PostMark [90], which simulates the workloads from emails, news, and Web commence applications. It starts with the creation of a pool of directories and files (creation phase), then issues a mix of transactions, including create, delete, read, and append (transaction phase), and finally removes all the directories and files (deletion phase). In contrast to the uniform, sequential data accesses used in the IOzone experiment, the file system is randomly accessed by PostMark with a variety of data and metadata operations. In the experiment, the initial number of directories and files were 200 and 2000, the file sizes ranged from 512B to 50KB, and the number of transactions was set to 20000.

The execution times of PostMark's various phases and its total runtime on NFS and GVFS are shown in Figure 3-8. Compared to NFS, the total runtime on GVFS is

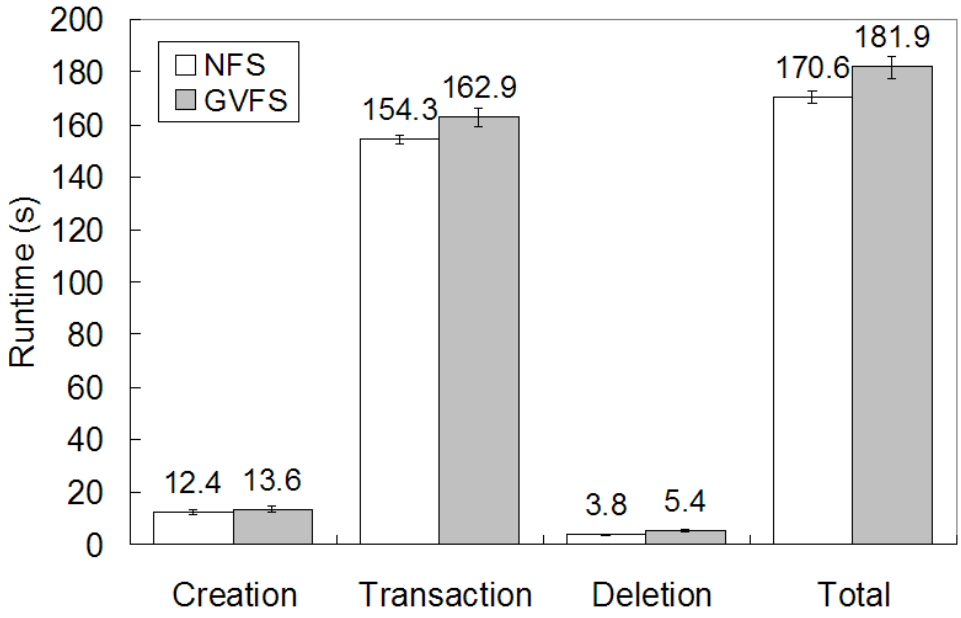

Figure 3-8. Runtimes of the various phases of PostMark as well as its total runtimes on NFS and GVFS.

only longer by 7%, and for the very intensive transaction phase, where a large volume of metadata and data updates are involved, GVFS is slower by 6%. It is evident that the overhead of GVFS' user-level virtualization is very small for such a benchmark that involves a large amount of disk accesses and exhibits a more typical data access pattern.

CHAPTER 4
APPLICATION-TAILORED DISTRIBUTED FILE SYSTEMS

User-level virtualization in GVFS provides the foundation for application-tailored DFSs. Based on virtualization, GVFSs can be managed on demand by middleware on a per-application, per-session basis (Figure 3-4): a GVFS is created before a compute session starts to provide remote data access for the application, and it is destroyed after the compute session completes. Such a data provisioning cycle is called a *GVFS session*. Since each GVFS session is dedicated to serving its application's remote data access, it can be customized with configurations that are tailored to the application's needs. Concurrent GVFS sessions can share the underlying physical software and hardware resources, whereas the virtualization layer enforces the isolation among them and allows them to be created and customized independently.

This virtualization also allows GVFS sessions to employ extensions and improvements that are not available in the physical DFS, and address the limitations and inefficiencies of the physical DFS. This chapter introduces such enhancements that are designed for application-tailored data provisioning, particularly in a wide-area, cross-domain environment. They cover various important aspects of DFSs, including performance, consistency, security, and fault-tolerance. Note that optimizations on these aspects (and others such as cost) cannot be considered in an isolated manner, because one often has implications on others. Therefore, tradeoffs have to be made to balance among these different goals. The flexibility provided by GVFS allows individual GVFS sessions to choose the configurations that best suit their needs.

4.1 Motivating Examples

While application transparency is a strong asset of DFS-based data provisioning approaches, it can also become a liability when DFSs are scaled to grid-style environments. The nature of WAN and grid resources decides that optimizations must be made for such environments in order to provide application-desired performance, consistency,

security, and reliability. However, it is often the case that "one size does not fit all". The enhancements need to be customized according to the data access characteristics and requirements, and considered in a context where application-specific modifications are unlikely to be implemented in kernels. These motivate the pursuit of user-level application-tailored DFS enhancements in this chapter. Potential uses of such enhancements can be illustrated with the following concrete scenarios.

Distributed virtual machines: Virtual machines (VMs) are increasingly used in distributed systems [17][79]. Efficient provisioning of VM images, which are typically very large in size, is key to dynamic instantiations of VMs across networks. Using DFS for remote VM state access allows distributed VMs to be quickly started without entirely transferring the large VM state files, and it supports many VM instances to be created from a small set of templates by read-only sharing their templates with independent copy-on-write state. Due to the absence of write sharing, both reads and writes can be cached (with write delay) on the client side to support efficient executions of the instantiated VMs, where a customized caching scheme is necessary to provide the capacity and persistence needed for the aggressive caching.

Software repositories: Software repositories are popular in enterprises as a means of sharing software among users. Such repositories are often set up on a DFS in an enterprise-scale network, read-only shared by organization users, and centrally managed by system administrators. However, as the scale of the enterprise's resources and users grows, support for wide-area sharing becomes a challenge to traditional DFS technologies. In this example, client-side data caching is important to improving user-perceived performance of using applications from the repository, but a cache consistency protocol is also needed to let the users see the latest software after it is updated by the administrator.

Scientific data processing: Scientific data are often continuously produced on-site and at the same time processed off-site in computing facilities. A DFS helps the distributed programs to conveniently share the possibly massive amount of data,

and it allows the analysis to be performed over different data ranges and with different granularity [91]. This scenario precludes the use of write delay on the data producing side, but permits reads to be cached on the processing side to speed up the analysis. The cache consistency protocol needs to support effective use of the cached data with small consistency maintenance overhead. Meanwhile, it should still provide a consistency guarantee that allows the generated data to be available for processing in a timely fashion.

GSI-enabled grid file systems: Employing DFSs to provide data to grid applications allows unmodified applications to transparently utilize computing and data resources across administrative domains. Security is critical in such grid file systems because data are shared among organizations with limited mutual-trust, and stored and transferred on resources with limited security. It is necessary to enhance the DFSs to support strong authentication, privacy, and integrity. These mechanisms also need to be compatible with the widely adopted grid security infrastructure (GSI [51]), so that the data management can be interoperable with other grid middleware and integrated with existing grid systems.

Long-running computation tasks: Large computation tasks, such as simulation and data mining, are often conducted in parallel on computing resources aggregated across LAN and WAN. Using DFSs to support these tasks enables the parallel processes to transparently share the inputs and outputs without explicitly transferring them. These tasks often take a long time, possibly days or even weeks, to finish, and thus require highly reliable executions. Their DFSs need to be tailored to provide good data availability that can tolerate failures happened on clients, servers, and networks. It is also desirable that they be able to automatically detect and recover from the failures, and support continuous remote data access for these tasks transparently.

To satisfy the diverse needs of applications, such as the ones in the above examples, the rest of this chapter presents the application-tailored enhancements on several important aspects of DFSs, including performance, consistency, security, and reliability.

4.2 Performance

Performance is one of the main hurdles preventing conventional DFSs to scale in wide-area environments. Those design decisions made under the assumption of a LAN-speed connection between client and server do not apply to WAN, where the round-trip latencies are often larger by orders of magnitude. Consequently, inefficiency appears when such DFSs are used in wide area, e.g., excessive interactions between client and server on a long-latency network link can cause significant increase on the response time and reduction on the throughput.

4.2.1 Client-Side Disk Caching

One particular limitation of conventional DFSs is on the use of client-side caching. Caching is a classic and successful technique to improve the performance of computer systems by exploiting temporal and spatial locality of data references and providing high-bandwidth, low-latency access to cached data. However, typical DFS clients employ only memory caching, because they assume that servers are in close proximity. Nonetheless, memory often does not have sufficient capability to exploit locality, and it is non-persistent and thus unable to support extensive write delay. For example, the NFS protocol allows the results of various NFS requests to be cached by an NFS client [21]. Although memory caching is generally employed by NFS clients, disk caching is not typical.

On wide-area file systems, caching is key to hiding long network latencies and improving an application's or user's data access experience, because the overhead of a network transaction is much higher compared to that of a local I/O access. Hence, GVFS provides persistent caching on client-side local disks to enhance the performance of remote data access on WAN.

4.2.1.1 Design

The GVFS disk caching effectively complements the existing memory caches in the native DFS. Its greater disk capacity promises reduction on cache capacity and

conflict misses [92] and thus less high-latency network communications. Its use of nonvolatile storage also allows more aggressive write-back caching because the delayed data modifications can be recovered across client restarts or crashes. Therefore, employing disk caching can form an effective cache hierarchy: memory is used as a small but fast first-level cache, whereas disk works as a relatively slower but much larger second-level cache.

Disk caching in GVFS is implemented at user-level by the file system proxy. A virtual DFS can be established by a chain of proxies, where the native O/S client-side proxy (the proxy client) can be employed to establish and manage disk caches. As illustrated in Figure 4-1, kernel buffer misses can be satisfied locally if they hit in the disk caches; otherwise they are forwarded to the server and the returned results are stored in the disk caches. The GVFS disk caching operates at the granularity of NFS remote procedure calls (RPCs). A GVFS disk cache is generally structured in a way similar to traditional block-based hardware designs: it contains file banks that hold frames in which data blocks and cache tags can be stored. Cache banks are created on the local disk by the proxy client on demand. The indexing of banks and frames is based on a hash of the requested NFS file handle[1] and offset and allows for associative lookups. The hashing function is designed to exploit spatial locality by mapping consecutive blocks of a file into consecutive sets of a cache bank, so that when a data block is serviced from the cache, its adjacent blocks can be quickly accessed from the cache as well. (More details about the cache design and implementation can be found in [93].)

The GVFS disk caching supports different policies for write operations: read-only, write-through, and write-back, which can be configured by middleware for specific user and application on a per GVFS session basis. Write-back caching is an important feature

[1] Files and directories are referred by file handles. A file handle is an opaque binary value which can be up to 64 bytes long in NFSv3 and even longer in NFSv4. A file's file handle is assigned by an NFS server, and it uniquely identifies the file on the server throughout its lifetime.

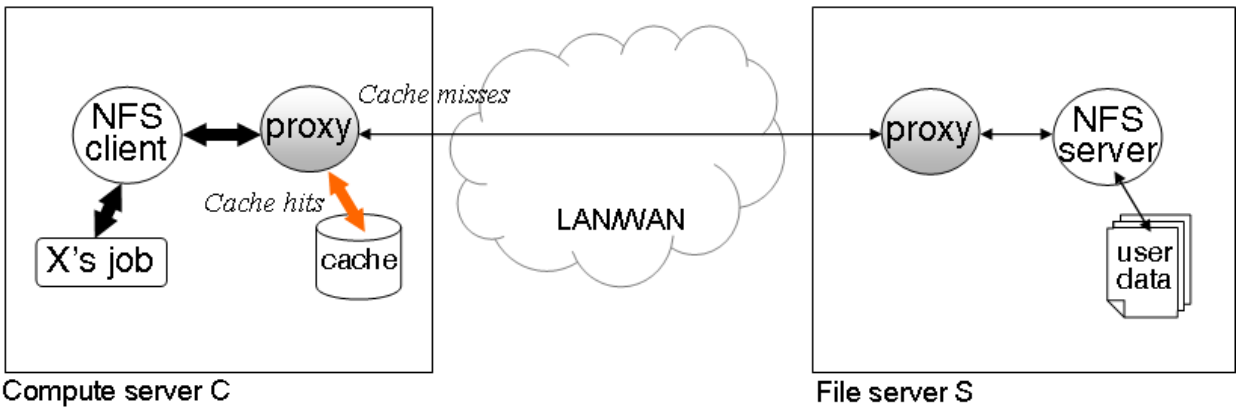

Figure 4-1. A GVFS-based virtual DFS can be established by a chain of proxies, where the native O/S client-side proxy can establish and manage disk caches. Kernel buffer misses can be satisfied locally if they hit in the disk caches; otherwise they are forwarded to the server and the returned results are stored in the disk caches.

in wide-area environments to hide long write latencies by leveraging the locality among write accesses. Furthermore, write-back disk caching can avoid transfer of temporary files. After the computing session completes, a user or data scheduler can remove temporary files from the working directory, which automatically triggers the proxy to invalidate cached modifications of those files. Thus when the proxy writes back the cached modifications, only the useful data are submitted to the server, so that both bandwidth and time can be effectively saved.

4.2.1.2 Deployment

As GVFS sessions are dynamically set up by middleware, disk caches are also dynamically created and managed by their proxy clients on per-session basis. When a GVFS session starts, its proxy client initializes the cache with middleware configured parameters, including cache path, size, associativity, and policies. During the session, some of the parameters, including cache write and consistency policies, can also be reconfigured. When the session finishes, policies implemented by grid middleware can drive the proxy to flush, write-back, or preserve cached contents as needed.

Typically, kernel-level NFS clients are geared towards a local-area environment and implement a write policy with support for staging writes for a limited time in kernel memory buffers. Kernel extensions to support more aggressive solutions, such as long-term, high-capacity write-back buffers are unlikely to be undertaken; NFS clients are not aware of the existence of other potential sharing clients, and thus maintaining consistency in this scenario is difficult. The write-back caching in GVFS can leverage middleware support to implement a session-based consistency model from a higher abstraction layer: it supports middleware to command a proxy client through O/S signals and control it to write back and flush cache contents.

Such middleware-driven consistency is sufficient to support many grid applications, e.g., when tasks are known to be independent by a scheduler for high-throughput computing. Furthermore, it is also possible to achieve fine-grained cache consistency models through inter-proxy coordination mechanisms, which are presented in Section 4.3.

4.2.1.3 Application-tailored configurations

There are several DFSs that exploit the advantages of disk caching too, for example, AFS [25] transfers and caches entire files in the client disk, and CacheFS [31] supports disk-based caching of NFS blocks. However, these designs require kernel support, and are not able to employ per-user or per-application caching configurations. In contrast, GVFS is unique in supporting customization on a per-user, per-application basis [94].

The GVFS sessions can employ disk caches independently from one another. The configurations of cache parameters (size, associativity) and policies (write-through, write-back) can be customized according to the data access patterns and requirements of applications. Specifically, for applications that use intensive writes, write-back caching can be employed to improve the application's data access performance; for read-mostly applications, the use of write-through can improve the tolerance of client-side failures and maintenance of data consistency.

The size of a GVFS session's disk cache can also be customized according to its application's needs. For example, when the use of storage is not free, the application can balance between performance and cost by adjusting its disk cache size. On the other hand, when the available storage capacity is limited, the middleware can allocate the disk space among the caches of concurrent GVFS sessions according to the resource utilization policy, e.g., based on the priority of applications, or based on the profits generated by hosting the applications. (See Section 6.2 for more discussions on policy-driven resource allocation for GVFS sessions.) Another concrete example of application-tailored disk caching is enabling heterogeneous disk caching using metadata handling and application-specific knowledge, in order to support block-based caching for virtual machine disk state and file-based caching for virtual machine memory state, as discussed in Chapter 5.

While caches of different GVFS sessions are normally independently configured and managed, GVFS also allows them to share read-only cached data for saving storage usage and exploring more data locality. Proxy clients running on the same host can access the shared caches directly. On the other hand, a series of proxies, with independent caches of different capacities, can be cascaded between client and server, supporting scalability to a multi-level disk cache hierarchy. For example, the proxy clients located in the same LAN can employ a dedicated cache server managed by an additional proxy that interposes between the proxy clients and servers, which forms a two-level hierarchy with GBytes of capacity in a client's local disk to exploit locality of data accesses from the node, and TBytes of capacity available from a LAN disk array server to exploit locality of data accesses from clients in the same LAN. Such a setup is studied in Section 5.4.3 to support fast virtual machine cloning.

4.2.1.4 Evaluation

This subsection presents the experimental evaluation of GVFS with disk caching using a prototype implemented based on the virtualization of NFS (v2 and v3). The evaluation focuses on the performance in WAN, the target environment of GVFS. For

easy deployment and control of the test bed, VMware-based virtual machines were used to set up the file system clients and servers, and a network emulator (NIST Net [95]) was employed to emulate the wide-area links among them. Each virtual machine was configured with 1 CPU and 512MB memory and was installed UBUNTU 7 with kernel 2.6.20. These virtual machines were hosted on a cluster, where each physical node has dual 2.4GHz hyper-threaded Xeon processors and 1.5GB memory. The system clock on a separate physical server was used to measure time, which suffices the granularity required by this evaluation.

This experiment compares NFS-based GVFS implementation with the native NFS. Unless otherwise noted, the version 3 of NFS over TCP was used for both. The servers exported the file system with write delay and synchronous access. The native NFS daemons used the default configuration of 8 threads, whereas the GVFS proxies were also multithreaded and use 8 worker threads. (See 4.2.2 for a detailed discussion on GVFS multithreading.) The data block size for read and write RPCs was set to 64KB. No swap was used on the physical and virtual machines during the experiments. Every run was started with cold kernel buffer and disk caches, if used, by unmounting the file system and flushing the disk cache.

The experiment evaluates the throughput of GVFS with a typical file system benchmark, IOzone [88]. It was executed in the read/reread mode, which sequentially reads a 512MB file twice from the server. Since the client and server have only 512MB of memory, the buffer cache does not help with its LRU-based replacement for the benchmark's sequential reads. This experiment is designed to study the overhead of virtualization in wide-area environments with the read phase, and demonstrate the benefits of disk caching with the reread phase. On NFS, both phases need to fetch the file entirely across the network from the server's disk. With the disk cache's greater capacity, GVFS can satisfy the reread phase's data access locally without contacting the server.

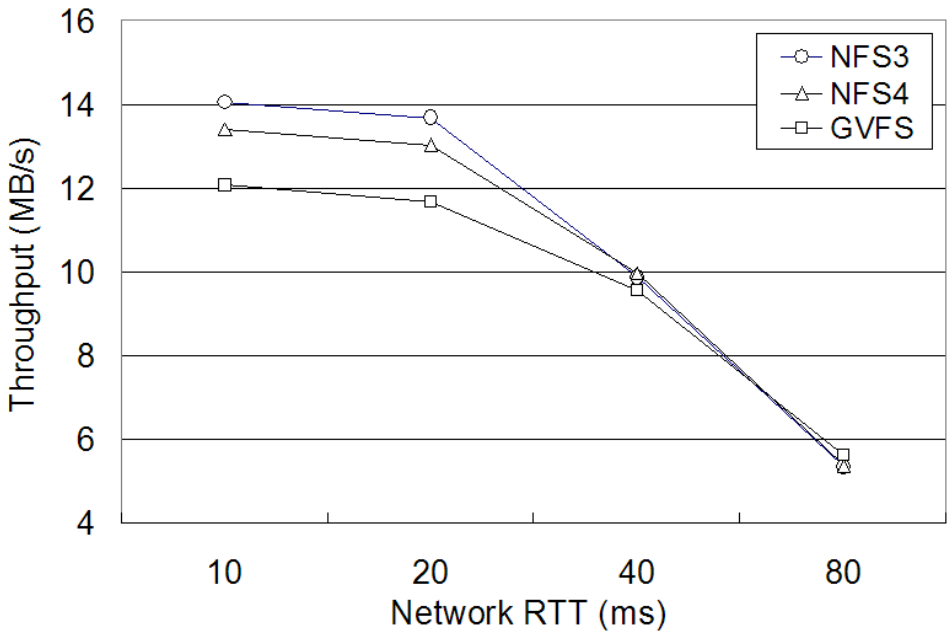

Figure 4-2. Throughputs of IOzone's read phase on NFSv3, NFSv4, and GVFS, with different network latencies between the client and server. (The standard deviations are all under 10% of the reported means.)

The throughputs of the read phase on NFSv3, NFSv4 and GVFS are compared in Figure 4-2, with different route-trip time (RTT) between the client and server. When the network latency is relatively small (less than 20ms), GVFS' throughput is 16% less than NFS. Recall that in the LAN experiment (discussed in Section 3.2.3), where the RTT is 0.071ms, the slowdown of GVFS is 25%. The longer network latency effectively diminishes the latency from the user-level virtualization and brings GVFS performance closer to NFS. When the RTT is beyond 40ms, GVFS behaves as well as NFS.

On the other hand, GVFS substantially outperforms NFS by leveraging data locality with disk caching. This is demonstrated by the throughput of the reread phase shown in Figure 4-3. With the help of warm disk caches, the throughput on GVFS is not affected by the growing network latency, and in fact, it is only bounded by the bandwidth of the client's local disks. Consequently, the speedup with respect to NFS increases as the network latency grows, and GVFS is four times faster when the RTT reaches 80 ms.

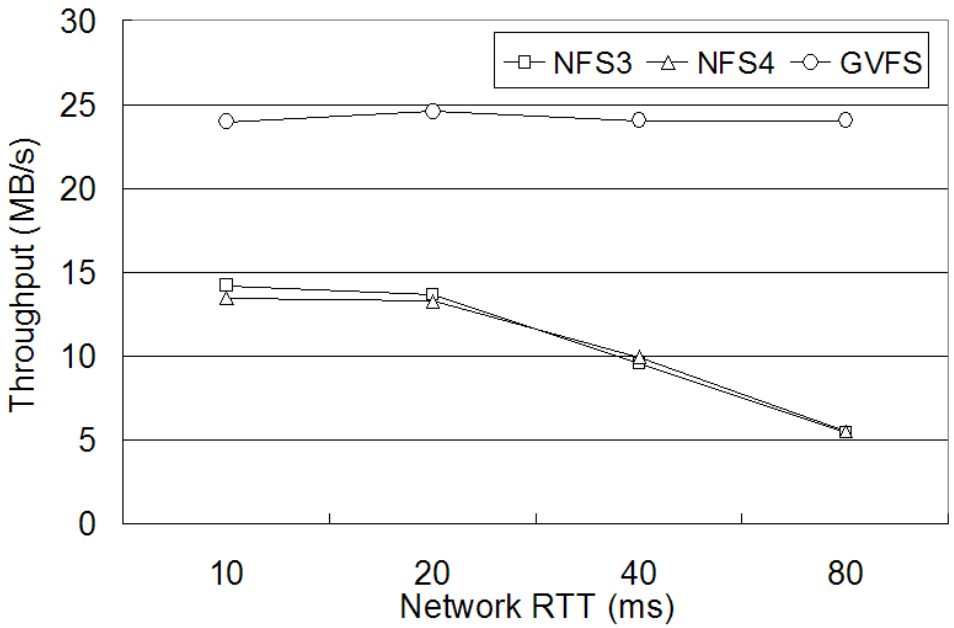

Figure 4-3. Throughputs of IOzone's reread phase on NFSv3, NFSv4, and GVFS, with warm caches, with different network latencies between the client and server. (The standard deviations are all under 10% of the reported means.)

4.2.2 Multithreaded Data Transfer

4.2.2.1 Design and implementation

The ability of overlapping data request processing and data block transfer is important to the throughput of remote data access. Conventional DFSs often use multiple daemons to serve incoming data requests, so that multiple outstanding data requests can be handled at the same time while waiting for the data accesses to complete on disks. For example, on a typical NFS server, 6 to 8 NFS daemons are usually running to serve the RPC requests from clients. This ability is even critical in a wide-area environment where the network latency is very high but the network bandwidth is sufficient to transfer data for multiple requests. If at any give time only one data request could be sent across the network, the remote data access would be significantly slowed down even though the network bandwidth is highly underutilized. Therefore, while virtualizing a conventional DFS such as NFS, the GVFS proxies need to be able to service data requests in a non-blocking manner in order to improve the remote data access throughput.

To achieve this goal, the GVFS proxy is enhanced by making it capable of multithreading. Specifically, the GVFS' NFS daemon consists of multiple threads which work around a RPC queue. A dispatcher thread is responsible for receiving RPC requests from the client and putting them in the queue. The other worker threads concurrently retrieve the requests from the queue, send them out to the server, and return the results to the client. In this way, even though every worker thread can only handle a single RPC request in a blocking manner, the entire proxy is processing the requests in a non-blocking manner.

The prototype of multithreaded GVFS proxy is developed on Linux, and its implementation is not trivial due to the fact that the standard RPC library is not multithreading-safe (MT-safe). Programs that make use of RPC on Linux typically utilize the existing RPC library provided by the standard C library, but the program cannot work correctly if it uses multiple threads to issue and service RPC requests. (This is caused by the fact that certain data handling structures in the RPC library is shared and thus conflicts happen when multiple threads are accessing it concurrently.) To address this problem, the prototype uses the TI-RPC library [96] to provide the RPC functionality. This library is an improved version of the existing RPC library in Linux, which provides generic RPC functionality to applications independently of the underlying transport protocols. (The existing library is transport-dependent and will eventually be replaced by the TI-RPC library.) Note that the TI-RPC library is still not completely MT-safe, and the proxy employs techniques to further improve upon that and make itself MT-safe (by replicating the shared data processing structures in the proxy to make sure that each thread has its dedicated copy to work with).

The use of multithreading in GVFS proxy also allows a GVFS session to customize its bandwidth usage according to its needs. Although the native NFS server is also multithreaded and can serve multiple clients' data accesses at the same time, it is not possible to isolate them from each other and control the bandwidth consumed by each client. In contrast, with multithreaded proxy, a GVFS session's bandwidth usage can be

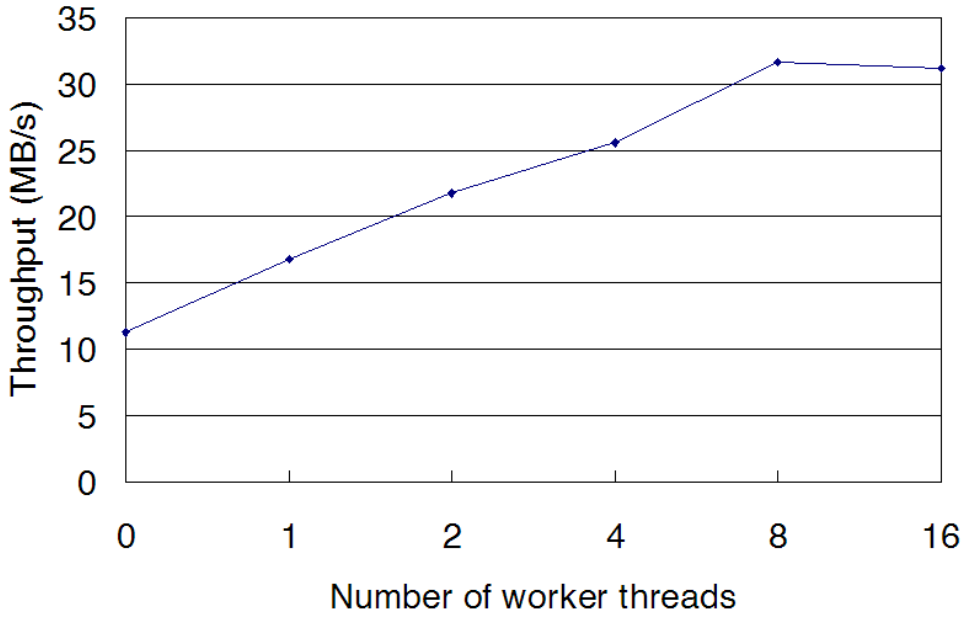

Figure 4-4. Throughput of IOzone with the number of GVFS worker threads varying from 0 to 16. The dispatcher thread is responsible for queuing the incoming RPC requests, whereas the worker threads are responsible for issuing the queued RPC requests to the server. When the number of worker threads is 0, the proxy is in fact not multithreaded and it blocks on every RPC request until the remote call is completed.

flexibly controlled by tuning the number of threads used by the proxy. As discussed in the customization of cache size in Section 4.2.1.3, the ability of controlling a GVFS session's bandwidth usage is important in two folds. First, it is important for an application to trade data access throughput for other considerations, e.g., cost, when the application has to pay for the resource usage. Second, it is necessary to allocate the shared network bandwidth resource among the concurrent DFSs based on policies so that the resource provider can optimize its resource provisioning.

4.2.2.2 Evaluation

This subsection uses an experiment to demonstrate the efficiency of multithreaded data transfer and the effectiveness of using the number of threads to control the bandwidth usage. The experiments was conducted on a Gigabit Ethernet where the file system client and server were set up on two virtual machines. Each virtual machines

was configured with 1 CPU and 512MB memory, and was installed with UBUNTU 7 with kernel 2.6.20. They were hosted on two physical servers, where each has 3.0GHz Pentium D processor with 4GB of memory. The experiment evaluates the throughput of GVFS with a typical file system benchmark, IOzone [88]. The benchmark was executed in the read mode, which sequentially read a 512MB file from the server.

Figure 4-4 compare the throughputs of IOzone with the number of GVFS worker threads varying from 0 to 16. The dispatcher thread is responsible for queuing the incoming RPC requests, whereas the worker threads are responsible for issuing the queued RPC requests to the server. When the number of worker threads is 0, the proxy is in fact not multithreaded and it blocks on every RPC request until the remote call is completed. The results show that the throughput grows practically linearly as the number of worker threads increases up to 8. However, when the number of worker threads goes beyond 8, the throughput slightly decreases. This is because 8 worker threads are sufficient to handle the incoming requests and sustain the maximum throughput in this setup, whereas more threads only causes more overhead from multithreading and degrades the overall performance. Nonetheless, these results prove that controlling the number of threads (between 0 and the number needed for the maximum throughput) can be an effective way to throttle the throughput of a GVFS session.

4.3 Consistency

As DFS clients widely employ local caches for performance improvement, cache consistency becomes a problem in that stale data may be seen by a client while another client has it modified. This happens when a client reads a stale copy of the data from its cache or when a client does not propagate its modified copy of data in a timely manner. To address this problem, a DFS needs to provide a proper cache consistency semantics to the clients in order to support the concurrent data sharing and deliver the application-desired data access behaviors. As discussed in Section 2.3.2, the definition of consistency in this dissertation only considers the order of read and write operations on

a single data item (e.g., a file). Although it is a weaker form of consistency compared to the models such as sequential consistency that specify constraints on the ordering with respect to the entire data set, it is sufficient to satisfy the needs of many applications. On the other hand, for applications that do require the stronger form of consistency, GVFS supports the use of file locking mechanisms to achieve that.

4.3.1 Architecture

The choice of a consistency model in a DFS is an important and difficult one, because it has implications in the complexity of developing applications (and the DFS itself) and in the performance of applications. A relaxed model may be acceptable (and desirable) to a simulation application, but may fall short of supporting database applications that rely on locks. A complex protocol that implements strong consistency may be desirable if it delivers high performance, but undesirable if it is difficult to implement, test, and deploy in existing O/Ss or if it requires applications to use a consistency-aware API.

A cache consistency protocol describes the implementation of a specific consistency model. For wide-area DFSs, such a protocol is important not only to the correct execution of the application, but also to its data access performance, because the client-sever interactions needed to maintain the consistency according to the protocol are very expensive on WAN. If DFSs are capable of leveraging application knowledge, the number of network transactions can be reduced, thereby reducing server loads and average request latencies.

The architecture described in this dissertation enables applications to use consistency protocols better suited than those native to a DFS in a manner transparent to the kernel and applications. In this architecture, illustrated in Figure 4-5, different consistency protocols can be overlaid upon the native DFS consistency mechanisms, and be applied to data sessions selectively and independently, based on the virtualization in GVFS. For example, native NFS protocols (v2 and v3) mainly rely on client-initiated revalidation requests to check for consistency. A proxy client hides the kernel client's consistency

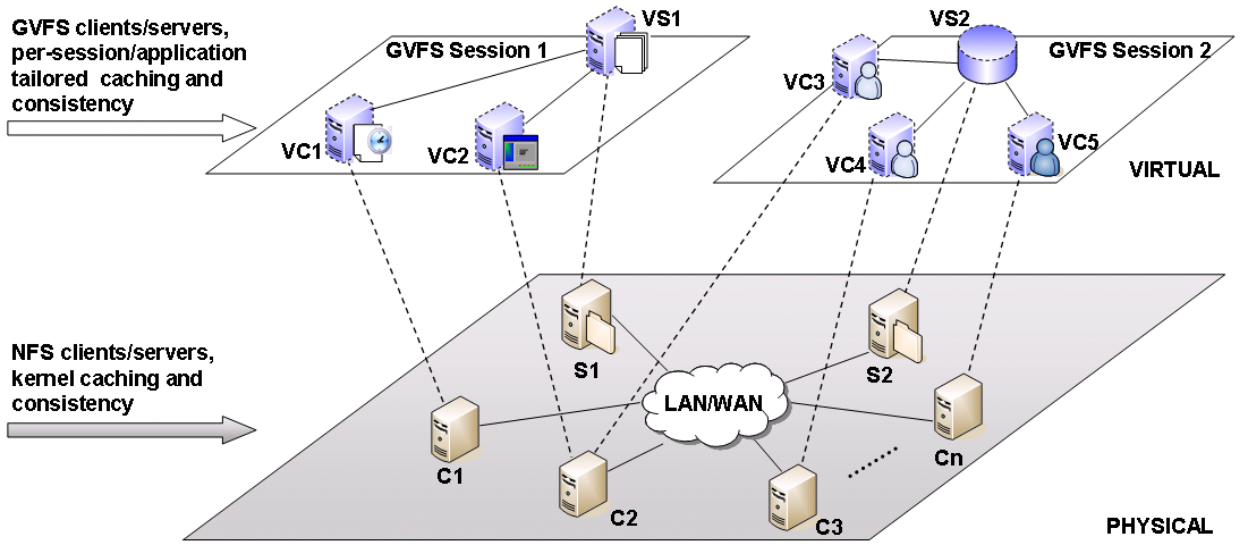

Figure 4-5. Application-tailored cache consistency protocols on GVFS sessions. The sessions consist of virtual clients (*VC1*-*VC5*) and servers (*VS1*-*VS3*) implemented by user-level proxies. They are dynamically established and managed by middleware and are overlaid upon shared physical resources (*C1*-*Cn*, *S1*-*S2*). Each GVFS session can employ independent application tailored user-level disk caching and consistency protocol. E.g., session 1 applies the delegation-callback based protocol (Section 4.3.3) and supports a scenario where real-time data are collected on-site (*VC1*) and processed off-site (*VC2*); Session 2 uses the invalidation-polling protocol (Section 4.3.2) to enable read-only sharing of a software repository (*VS2*) among WAN users (*VC3*, *VC4*) and maintenance update by LAN administrator (*VC5*).

checks from the data session by serving them locally, and it instead uses the user-level mechanisms to cooperate with the proxy server to keep data consistent across the network. In this way, kernel clients and servers are oblivious to these user-level protocols; the GVFS proxies, however, can be configured to maintain the consistency for the data sessions according to their selected protocols.

A variety of protocols are supported in GVFS, including the underlying DFS consistency itself, and more importantly, alternative ones that are specially designed for wide-area environments. These customized protocols are explained in detail in the rest of this section. They can achieve different levels of consistency, and can be selected and tuned by middleware based on the application needs on a per-session, per-application

Configuration File

base_path	/home/cache
session_key	XXYYZZ
acache_enabled	1
dcache_enabled	1
wb_enabled	1
inval_enabled	1
acache_size	65536
acache_asso	8
acache_banks	128
dcache_size	1048576
dcache_asso	16
dcache_banks	512
inval_min	3
inval_max	60

Figure 4-6. Sample configuration file used to customize a GVFS proxy cache as well as the consistency protocol. The parameters include the path and session ID of the proxy cache, the size, associativity and bank numbers of the attribute cache (*acache*), data cache (*dcache*), and the use of write-back and invalidation-based consistency.

basis. Figure 4-5 illustrates two GVFS sessions that are customized to support data provision for two example applications described in Section 4.1.

Two components are key to realizing GVFS-based application-tailored cache consistency over a wide-area environment: 1) file system proxies providing per-application GVFS sessions and enhanced with user-level disk caching and consistency; and 2) a middleware-based service that schedules the GVFS sessions and configures their use of caching and consistency according to application needs. Figure 4-6 shows an example of a configuration file used to customize a proxy cache as well as the consistency protocol. The parameters include the path and session ID of the proxy cache, the size, associativity, and bank numbers of the attribute cache (*acache*) and data cache (*dcache*), and the use of write-back and invalidation-based consistency. Such a configuration file is used by middleware to establish an application-tailored GVFS session. This chapter focuses on

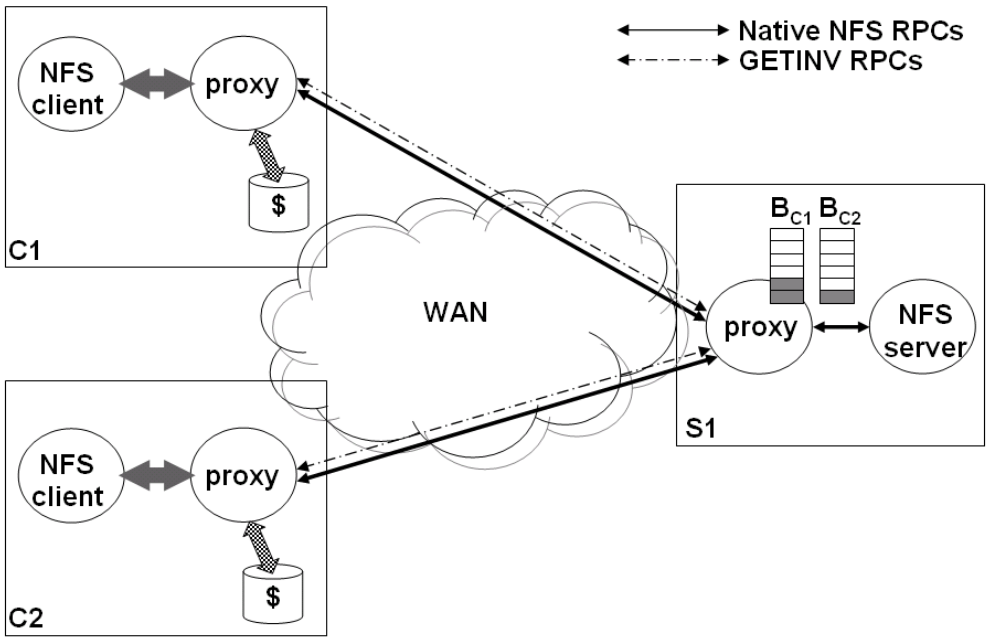

Figure 4-7. A GVFS session using the invalidation polling consistency protocol between the proxy clients at *C1*, *C2* and the proxy server at *S1*. The RPCs issued from kernel NFS clients can be served from the disk caches, while the proxy clients poll the proxy server for contents of per-client invalidation buffers (*BC1*, *BC2*) to maintain consistency.

the core mechanisms supporting the first component. The mechanisms to schedule and configure GVFS sessions on demand will be investigated in Section 6.1.2.

4.3.2 Invalidation Polling Based Cache Consistency

4.3.2.1 Protocol

This protocol employs invalidation buffers that reflect potential modifications to many files to reduce the rate at which per-file information is polled. Such an approach proves effective when modifications to the file system are infrequent and need to be quickly propagated to clients. The approach is illustrated in Figure 4-7: the proxy server of a GVFS session keeps track of logically timestamped file handles that need to be invalidated in per-client buffers; the proxy clients use a new protocol message — GETINV — to request information related to the invalidation buffer.

Server-side: When the proxy server receives a file modification request from a client (e.g., CREATE, WRITE), it adds the file's NFS file handle into the other clients'

invalidation buffers, since this file needs to be invalidated from the other clients' caches later. These per-client invalidation buffers are of finite size and implemented as circular queues. Multiple invalidations to the same file in an invalidation buffer can be coalesced in order to save space. The timestamps associated with each invalidation entry are generated by the server and increased monotonically with incoming requests. A proxy client's GETINV request contains the timestamp of the last invalidation it has performed, and the proxy server returns the file handles stored in the client's invalidation buffer, which represent the files that the client needs to invalidate in its cache. The proxy server also returns its current, updated timestamp to the proxy client, which will be used in the client's next GETINV request. The server can handle protocol cases where invalidation information is not fully available by using a flag (*force-invalidate*) to inform the client to invalidate its entire attribute cache. The proxy server processes a GETINV call as follows:

1. If this is the first GETINV call received from the client: initialize an invalidation buffer for the client, return updated timestamp and force-invalidation flag with value 1. Else,

2. If the timestamp in the GETINV request is earlier than the earliest one in the client's invalidation buffer: flush buffer and return updated timestamp and force-invalidation flag with value 1. Else,

3. Return buffer contents (and clear them), updated timestamp, and force-invalidation flag with value 0. If the buffer contents do not fit in a single RPC message, then return a poll-again flag with value 1 along with partial buffer contents.

Client-side: The proxy client polls the server with GETINV calls for potential invalidations occurred since its last known timestamp within a short time window. The polling time window can be fixed or varied within a configurable range using an "exponential back-off" style policy — the window size doubles every time when there is no invalidations during the previous window, until it reaches the maximum value, and it drops to the minimum value if invalidations have happened during the previous window. The received invalidations are performed by invalidating the cached attributes of the

concerned files, which will cause the proxy client to revalidate these files when they are accessed again. The proxy client processes the result from the GETINV call as follows:

1. Update a local variable holding the last known server timestamp.
2. If force-invalidation is equal to 1: invalidate its entire attribute cache. Else,
3. Scan the returned buffer and invalidate the attributes of the concerned files in its cache. And,
4. If poll-again is equal to 1: send another GETINV call to the server immediately.

In summary, only the file modifications observed by the proxy server cause invalidations on the proxy clients and they are transferred in a small number of GETINV replies. Only the files that are modified by the other clients during the past polling time window need to be revalidated by a proxy client, but all the other per-file consistency checks issued from the kernel NFS client will be filtered out during the next time window.

4.3.2.2 Bootstraping

The protocol uses logical timestamps to manage invalidation buffers. These are created by proxy server and used as arguments to GETINV calls by proxy client. The bootstrapping mechanism that provides an initial timestamp to a client uses a GETINV call with a null argument. Another form of bootstrapping takes place if the server fails or restarts and loses timestamp information. In this case, a client has a timestamp which is invalid and must obtain a new, valid timestamp. The server handles this case by returning a new timestamp and a force-invalidation flag to each client's first GETINV after it comes back.

4.3.2.3 Failure handling

The main factor that facilitates failure handling in this protocol is that the state stored on proxy server (invalidation buffers, timestamps) and clients (cached attributes and timestamps) is soft state which can be safely discarded. If the server crashes, once recovered it can initialize new invalidation buffers from scratch, bootstrap clients with new timestamps as described above, and continue to serve the clients' GETINV calls.

If a client crashes and loses its timestamp, then after recovery it issues GETINV with a null argument, and the server returns the latest timestamp with a force-invalidation flag. The same mechanism can be used if the client implements a policy to limit the number of invalidations it should process and bound the overhead from performing the individual invalidations, effectively allowing the client to force a self-invalidation on its entire attribute cache.

If a network partition happens, it is possible that the server's invalidation circular queue for the client has wrapped-around when it receives a GETINV from the client again. The server can detect this case by comparing the client's current timestamp with the earliest timestamp in its buffer. If the former one is earlier, it means that the client has not kept up with the invalidations, so the server should return the force-invalidation flag and an updated timestamp.

The invalidation mechanism is intended to provide relaxed consistency for the benefit of performance, but inconsistency can occur during the polling time window: a client may read a stale data block or file handle. It is appropriate for applications that can tolerate modest inconsistency (with the help from user or middleware). In practice, write sharing happens much less often than read sharing. Therefore this protocol is capable of providing applications with good performance and acceptable consistency. However, if stronger consistency is required, the delegation callback based protocol described below is better suited.

4.3.3 Delegation Callback Based Cache Consistency

4.3.3.1 Delegation

Strong consistency can be achieved in GVFS via delegation and callback mechanisms. A delegation gives a proxy client the guarantee to perform operations on the cached data without consistency compromises, whereas callback is used by a proxy server to revoke the delegation in order to avoid potential conflicts. Delegation and callback decisions are made by the proxy server on per-file basis. A GVFS session can realize strong consistency by (1)

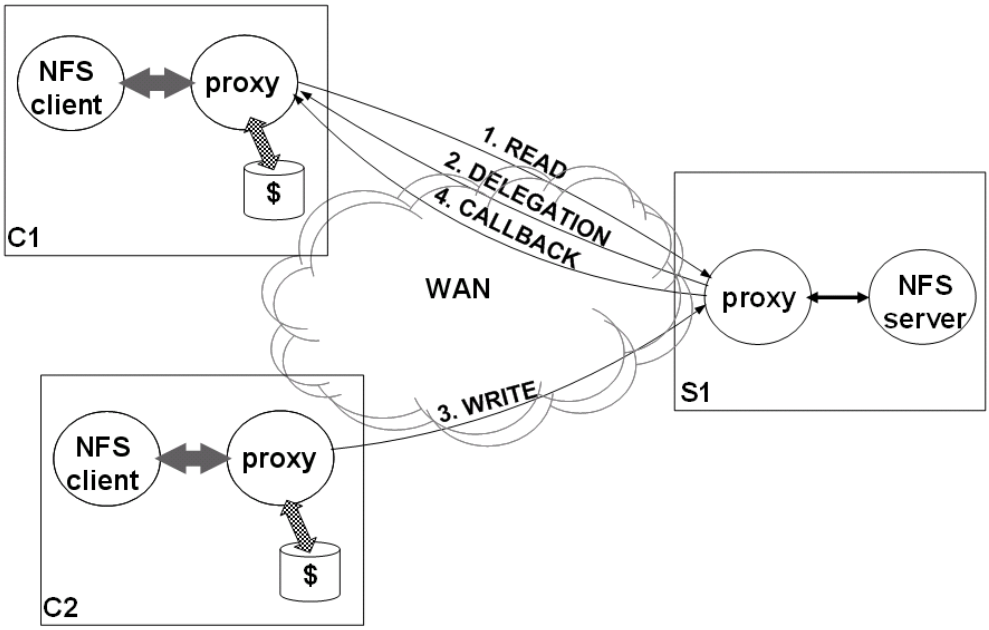

Figure 4-8. A GVFS session using the delegation callback based consistency protocol between the proxy clients at C1, C2 and the proxy server at S1. The figure shows the sequence of interactions happened during a read delegation and its callback.

disabling the kernel NFS client's attribute cache to force revalidations on every accessed file, and (2) enabling the GVFS cache's delegation callback protocol to handle consistency enforcement. Two types of delegations are provided. Read delegation allows a client to read cached data without revalidation; the periodic consistency checks issued by kernel NFS client can be fully handled at client side. With a write delegation, the proxy client can further delay writes; both read and write requests to the file can be satisfied from the GVFS cache without contacting the server.

In the absence of open and close file operations in NFS (v2, v3), a proxy server speculates about these operations by tracking a client's data access. When a read or write request is received the corresponding file is considered "opened" by the client. In a read sharing scenario, multiple clients can have read delegations on the same file at the same time. But write delegation can be granted only if no other clients have the file opened. When there is no sharing conflict a client obtains a delegation automatically with its first

read/write request for the file. Otherwise, the conflicting request triggers the proxy server to recall the file's existing delegations and make it temporarily non-cacheable until the conflict is resolved.

On the other hand, when a file has not been accessed by a client for a while, it is speculated as closed by the client and the proxy server issues callback if this client has a delegation on the file. To allow a client to automatically renew a delegation, the proxy client periodically let a request for the file bypass the cache. The delegation's expiration and renew periods are both configurable per session, e.g., 10 minutes and 8 minutes respectively. The callback ensures the correctness of consistency even if the clocks on the server and client are badly skewed.

For the above mentioned proxy server-to-client interactions, the delegation and cacheability decisions are either piggybacked on a native NFS reply message, or enclosed in the GVFS callback calls. Figure 4-8 shows an example of these interactions.

4.3.3.2 Callback

A callback requires a server-to-client RPC call, which is inherently supported in GVFS because a proxy works as both RPC client and server. A proxy client encapsulates its listening port number along with its identification in regular RPC requests, so the proxy server knows how to connect an authenticated client for callbacks. To avoid deadlocks, the proxies are multithreaded to serve both NFS RPCs and GVFS callbacks. Correspondingly, separated queues are also maintained to buffer these two types of calls.

Callback of a read delegation invalidates the file's attributes in the proxy client's cache, which causes revalidation of the file's cached data when they are accessed again, whereas callback of a write delegation also forces the write back of cached data modifications. In a simple implementation, the callback does not return until all the data have been submitted to the server. However, the volume of dirty data can be very large and thus the callback as well as the other client's request which triggers this callback have to be blocked for a long time and may eventually time out. Note that this is still safe

because both NFS and callback requests can be simply retried. But it is not desirable if the application which waits on the write-back perceives a substantial response delay.

Since a request to a single block does not have to wait for the entire file being written back, the protocol is optimized as follows. If the number of cached dirty blocks is considerably large (e.g., more than 1K blocks), the proxy client returns a list of these blocks' offsets for the received callback. The block that is requested by the other client is immediately written back (if it is indeed dirty), but the other blocks are submitted afterwards. (To realize this, the requested block's offset is sent along with the file's NFS file handle in the callback.) Upon receiving the block list and the first block, the proxy server considers the write delegation revoked. However it needs to monitor the progress of the write-back and update the list accordingly until it completes. Meanwhile, requests from other clients to the blocks that are not written back will still generate callbacks to force the client to submit them promptly.

4.3.3.3 State maintenance

The proxy server manages a GVFS session's state using a list for participating clients and a hash table for opened files. Client identification is provided by unique session key or distinguished name encapsulated by proxy client in every RPC request. (Please see Section 4.4 for details on authentication mechanisms.) The client list stores their IDs and callback ports. Each opened file has an entry in the hash table to record its current state and sharers' client IDs. A timestamp is also kept along with a client ID and updated every time the file is accessed by the client, which is used to speculate on the file close. Once the file is considered closed by a client, the client's information is removed from the file's entry; an entry is deleted from the hash table when the file is not opened by any client any more.

The expiration time determining whether a client has closed a file or not presents a tradeoff. Its value cannot be too small; otherwise delegations are given out too often which requires many callbacks to maintain the consistency. In contrast a long expiration time

snags a client from getting delegations and can potentially hurts its performance, while the proxy server also needs to track a large number of files and sharers. In the latter case, the proxy server can reduce the amount of state by proactively issuing callbacks on the least recently accessed files and then evicting their entries.

4.3.3.4 Failure handling

Failure handling is more important to this consistency protocol than the invalidation polling based protocol, because the state stored at the proxy server is crucial to strong consistency. But delegations also provide the proxy clients opportunities to continue serving applications' requests from locally cached data even in presence of server crash or network partition. After the server comes back it can reconstruct the session's state by issuing special callbacks to all the known participating clients. To realize this, the client list data structure mentioned above is always stored directly in disk.

This type of callback is different because it targets at the entire cache rather than a specific file. The read delegation holders will invalidate all the cached attributes and thus require revalidation of every cached file when it is reaccessed. A proxy client that has write delegations will also reply the callback with a list of locally modified files so that the proxy server can rebuild the hash table. Note that before every client has answered the callback, the proxy server should block all the incoming requests. However, this grace period is considerably short because it only requires a single multicasted callback to the clients. If the callback to a client times out, the client is assumed failed and not considered any more.

The nature of disk caching guarantees that a proxy client would not lose anything after it recovers from a failure and it can easily reconstruct the list of dirty blocks by scanning the entire cache once. However, it needs to contact the server to reconcile any inconsistency happened during its crash. Therefore, it invalidates the entire attribute cache to force revalidation on all the cached files. In addition, for a file that has modifications delayed in cache, the proxy client checks with the proxy server whether

the file is updated by other clients or not during the crash. If not, the proxy client then writes back the cached modifications; otherwise, those data are considered stale and are discarded. Note that if the user or middleware that manages the session makes sure that no write sharing happens during the client's crash, then no data loss will happen after the client is recovered. On the other hand, this protocol also allows the client/middleware to quickly give up on the failed client and redo the data operations on the session from another client.

4.3.4 Evaluation

4.3.4.1 Setup

The proposed GVFS cache consistency protocols are evaluated in this section using experiments on both microbenchmarks and application benchmarks. Microbenchmarks exercise GVFS with simple programs to demonstrate its performance compared to conventional DFSs, whereas application benchmarks use real scientific tools to investigate GVFS in typical grid computing scenarios.

The emphasis of the experiments is in wide-area environments, which were emulated using NIST Net [95]. Each link between the file system clients and server was configured with a typical wide-area RTT of 40ms and bandwidth of 4Mbps. Six file system clients and one file system server were set up on VMware-based virtual machines, which were hosted on two physical servers. Each physical server has dual 2.4GHz hyper-threaded Xeon processors and 1.5GB memory. Each virtual machine was configured with 256MB memory and was installed with SUSE Linux 9.2. The use of network emulator and VMs facilitates the quick deployment of a controllable, duplicable experimental setup. However, timekeeping within a virtual machine is often inaccurate, so the system clock on a physical server was used to measure time, which suffices the granularity required by this evaluation.

The experiments were mainly conducted on file systems mounted through native NFSv3 and NFSv3 based GVFS, both with ACL disabled. The file system server exported the file system with write delay and synchronous access. Every experiment was started

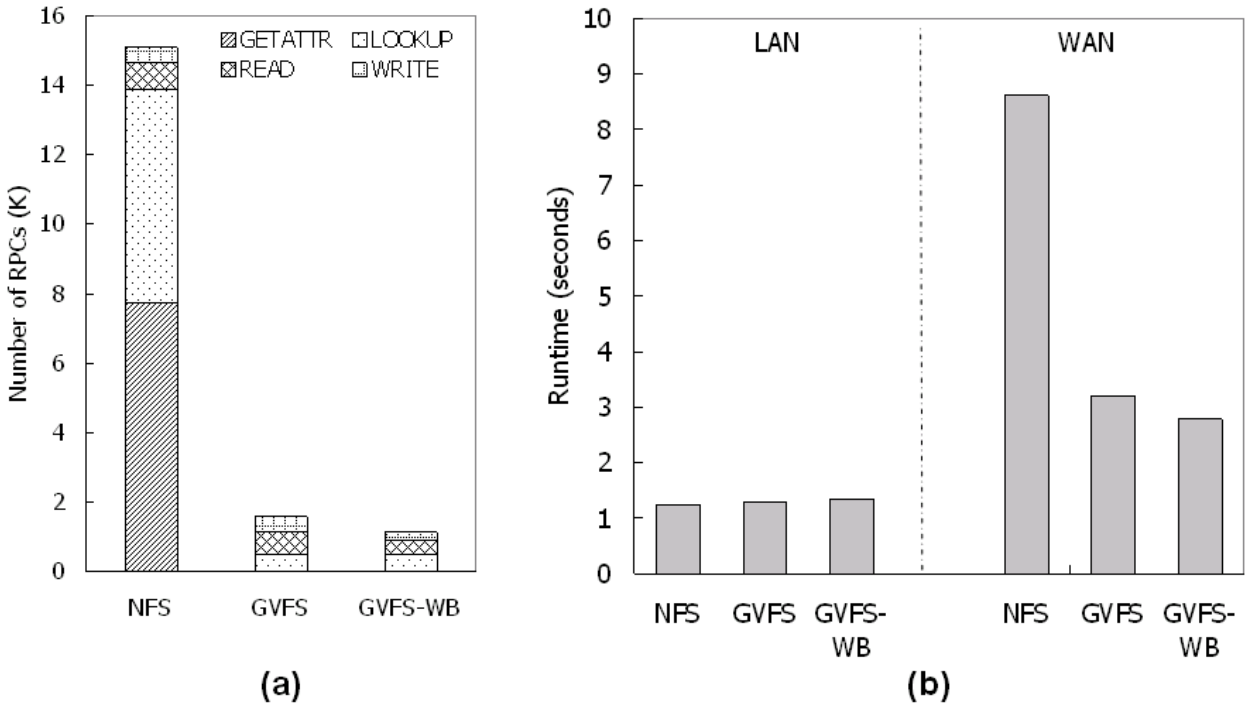

Figure 4-9. (a) Numbers of RPCs transferred over the network during the execution of the Make benchmark. (b) Runtimes of the Make benchmark. The benchmark was executed on different setups: NFS (*NFS*), GVFS with read-only disk caching (*GVFS*), and GVFS with write-back disk caching (*GVFS-WB*).

with cold kernel buffer and GVFS disk caches by unmounting the file system and flushing the disk cache.

4.3.4.2 Make

The first benchmark demonstrates the performance edge of GVFS caching in a single client scenario. It runs "make" on an application's source code (Tcl/Tk8.4.5), similar to the Andrew benchmark [25]. The make takes 357 C sources and 103 headers to generate 168 objects. It was executed on a file system mounted from the server via three different setups: native NFS (*NFS*), GVFS with read-only caching (*GVFS*), and GVFS with write-back caching (*GVFS-WB*). This benchmark mainly exercises the GETATTR, LOOKUP, READ, and WRITE RPCs. The execution times and RPC counts are reported in Figure 4-9.

The data from executions on NFS show that, although the make only accesses hundreds of files, it generates tens of thousands of cache consistency checks (GETATTR calls) in the process of cross-referencing existing files to produce new objects. However, the results from GVFS prove that the disk cache can virtually satisfy all of them with its consistency protocol. When the client uses invalidation-polling with a typical period (e.g., 30 seconds), only tens of GETINV calls are required; with delegation callback based consistency there are no extra calls. The larger capacity of the disk cache also substantially reduces the number of LOOKUP calls, and the use of write-back further decreases the number of READs and WRITEs. Consequently, in WAN environment GVFS runs the benchmark three times faster than NFS. The runtime of the benchmark in a 100Mbps LAN was also measured and is reported in the figure. It shows that GVFS is relatively slower than NFS in LAN due to the overhead from RPC interception and cache management. However, this overhead is very small: only 4% with read-only caching and 8% when write-back is also used, and as network latency grows it should be overcome by the gain from saving network trips, as confirmed by the next experiment.

4.3.4.3 PostMark

The second experiment uses the PostMark [90] benchmark which simulates the workloads from emails, news, and web commence applications. The experiment with PostMark was conducted on NFS and GVFS in different network environments by varying the end-to-end latency. Two GVFS setups that employ different cache consistency protocols are used: *GVFS-inv* uses the invalidation-polling protocol, overlaid upon the default kernel NFS cache configuration; and *GVFS-cb* disables the kernel attribute caching and applies the delegation-callback protocol. The benchmark was configured with the following parameters: the initial number of directories and files are 20 and 200, the file sizes range from 512B to 16KB, and the number of transactions is set to 2000.

The total execution times of PostMark on the above setups with different network latencies are compared in Figure 4-10, and the runtimes of the benchmark's various phases

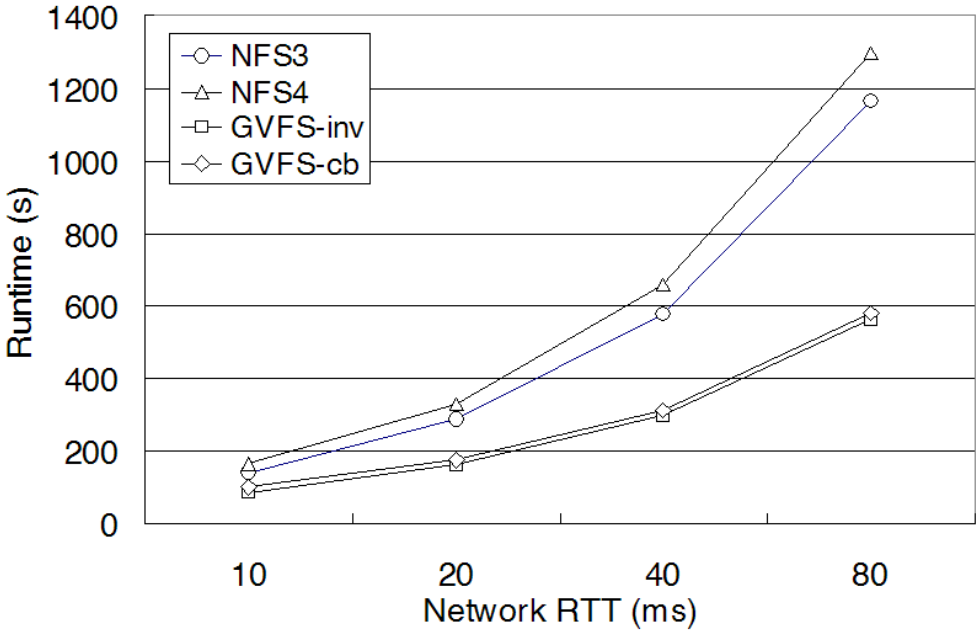

Figure 4-10. Runtimes of PostMark with different network latency over NFSv3 (*NFS3*), NFSv4 (*NFS4*), GVFS with default kernel buffer setup (*GVFS-inv*), and GVFS with kernel attribute caching disabled (*GVFS-cb*).

for the case of 80ms-RTT are plotted in Figure 4-11. The results show that both GVFS setups substantially outperform NFS in typical WAN environments, and the speedup is about 2-fold when the RTT is beyond 20ms. This can be attributed to the significant amount of client-server interactions saved by using GVFS cache consistency protocols. Specifically, when the RTT is 80ms, the number of RPCs received by the server during the execution on GVFS is 47% and 37% of that on NFS v3 and v4, respectively.

Because *GVFS-cb* has the kernel attribute caching disabled, it needs to handle more requests from the kernel client and thus it is relatively slower than *GVFS-inv*. This is the price it has to pay in order to realize strong consistency, but the slowdown is very small and its performance is still considerable better than native NFS protocols. Note that the NFSv4 implementation is still at the experimental stage, and future improvements may support more aggressive caching and deliver better performance. Nonetheless, such support will still be deeply embedded in the kernel, and it is unlikely to be tuned or modified according to the needs of specific applications.

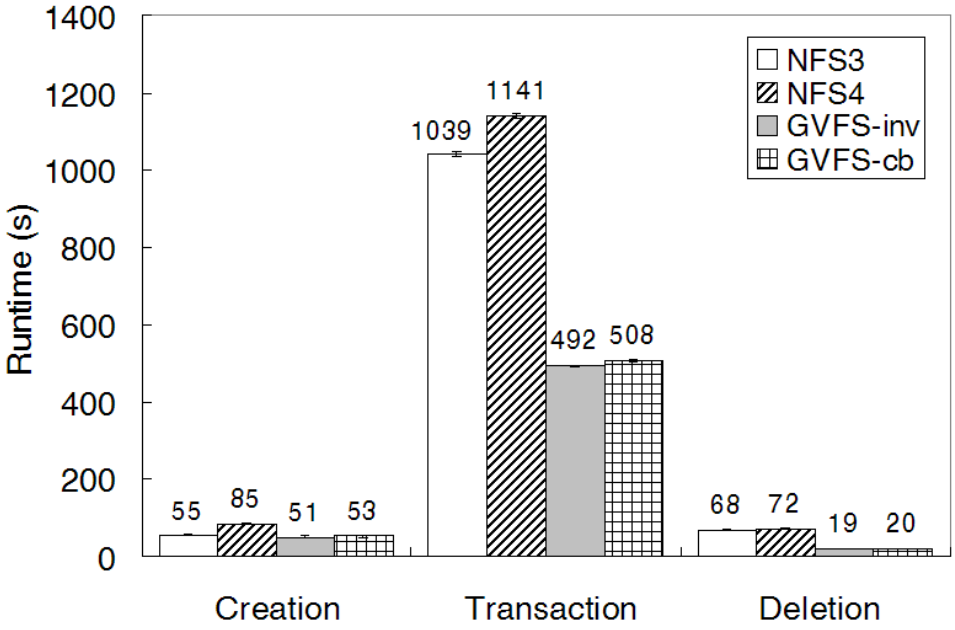

Figure 4-11. Runtimes of the different phases of the PostMark benchmark over NFSv3 (*NFS3*), NFSv4 (*NFS4*), GVFS with default kernel buffer setup (*GVFS-inv*), and GVFS with kernel attribute caching disabled (*GVFS-cb*). The network RTT is 80ms.

4.3.4.4 Lock

The third benchmark studies the behavior of GVFS' different consistency protocols when supporting cooperative, multiple-client workloads. It uses a popular mutual exclusion mechanism on file systems: file-based locks. In the experiment, six distributed clients compete for a lock by creating an independent temporary file and attempting to hard-link it to the shared lock file. If a client gets the lock, it pauses for a period of ten seconds and then releases the lock by unlinking the lock file. Otherwise, it pauses for a second and tries for the lock again till it gets the lock. After a client releases the lock, it also pauses for a second and then rejoins the competition till it succeeds for ten times.

This experiment was conducted in WAN with different consistency protocols. It serves as a good example of the tradeoff between consistency and performance. When consistency is relaxed, a client may not see the release of the lock immediately, and the previous owner of the lock tends to get it again. On the other hand, stronger consistency

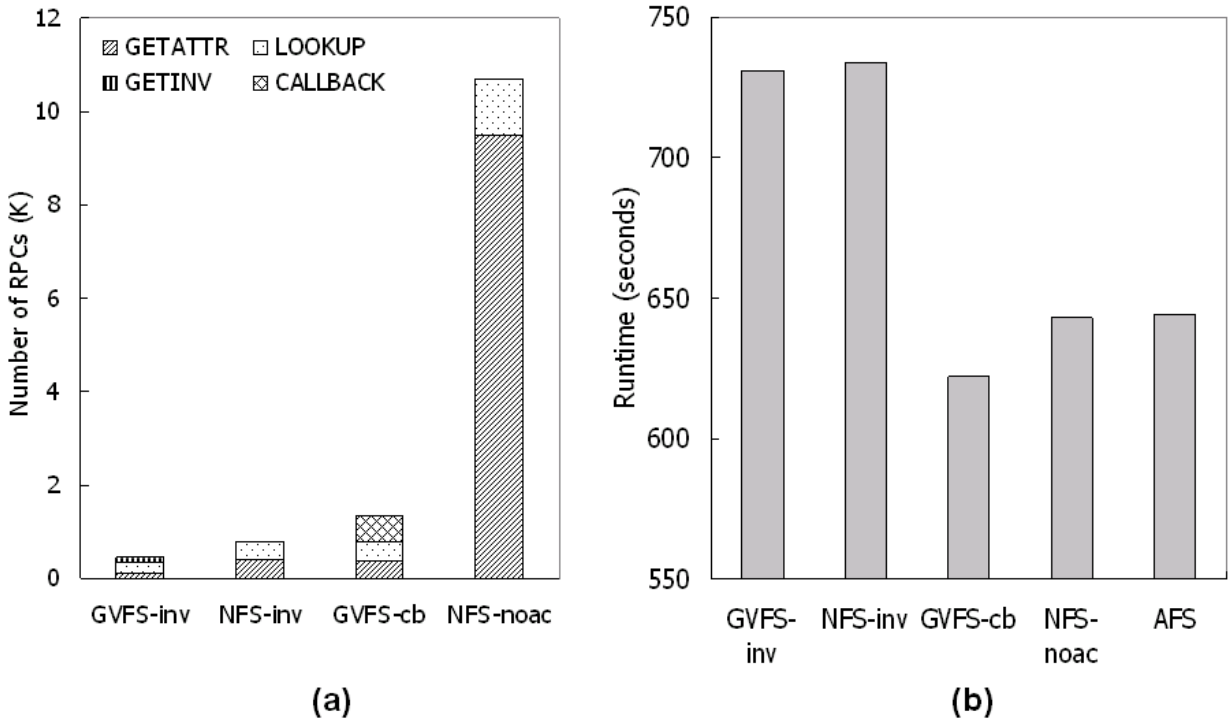

Figure 4-12. (a) Numbers of RPCs transferred over the network during the execution of the Lock benchmark. (b) Runtimes of the Lock benchmark. The benchmark was executed across WAN with different setups: NFS with 30s revalidation period (*NFS-inv*), NFS with no attribute cache (*NFS-noac*), GVFS with 30s invalidation period (*GVFS-inv*), GVFS with delegation and callback (*GVFS-cb*), and AFS. The RPC counts used by AFS are not shown because it uses a different RPC protocol and is not comparable.

provides better fairness among the clients but also consumes more bandwidth and generates higher server loads due to the use of more consistency calls. To demonstrate the first case, the benchmark was executed on NFS and GVFS both with a revalidation/invalidation period of 30 seconds (*NFS-inv* and *GVFS-inv*). For the second case, the experiment was conducted with NFS with no attribute cache (*NFS-noac*) and GVFS with delegation and callback (*GVFS-cb*).

By analyzing the distribution of lock acquired from the experimental results, it is confirmed that fairness can be achieved with the strong consistency models but not with the weak ones. Further, Figure 4-12 shows that, in the latter case the benchmark takes

nearly twice longer to execute, also because of the delay for a lock release being observed by other clients.

The overhead involved in achieving the same level of consistency is significantly different between NFS and GVFS. GVFS' client polling protocol uses 44% less consistency checks (GETATTR, GETINV) than NFS (GETATTR). In the stronger consistency case the difference between NFS and GVFS is even more dramatic. The consistency related calls issued by NFS (GETATTR) outnumbers that of GVFS (GETATTR, CALLBACK) by more than 10-fold. Hence substantial bandwidth and load are saved by using GVFS.

Note that although the benchmark runs faster on GVFS than on NFS, the advantage is not so large as in the number of RPCs. This is because most of the extra RPCs' latencies are overlapped with lock owners' pausing times during the execution.

As a reference, another traditional DFS that delivers strong consistency, AFS (OpenAFS 1.2.11 [25][26]), was also tested with the benchmark. The above experiments prove that GVFS can flexibly and efficiently provide different application-tailored consistency protocols, which is difficult to achieve with traditional DFSs.

4.3.4.5 Software repository

The wide-area shared software repository scenario discussed in Section 4.1 is studied here with NanoMOS, a 2-D n-MOSFET simulator. This is a compute-intensive application and benefits from parallel execution on grid resources. A wide-area file system supports it by allowing the WAN users to read-share the application and its required software, including MATLAB with the MPI toolbox (MPITB), and allowing the local administrator to maintain the repository at the same time. In the experiment, NanoMOS was stored along with the other software in the repository on the server, and it was executed in parallel on six clients for eight consecutive iterations. The repository was maintained by the administrator from another client, and between the fourth and fifth run a software update was performed on the repository. Two different cases of updates were considered: update to MPITB only and to the entire MATLAB package. The repository was shared

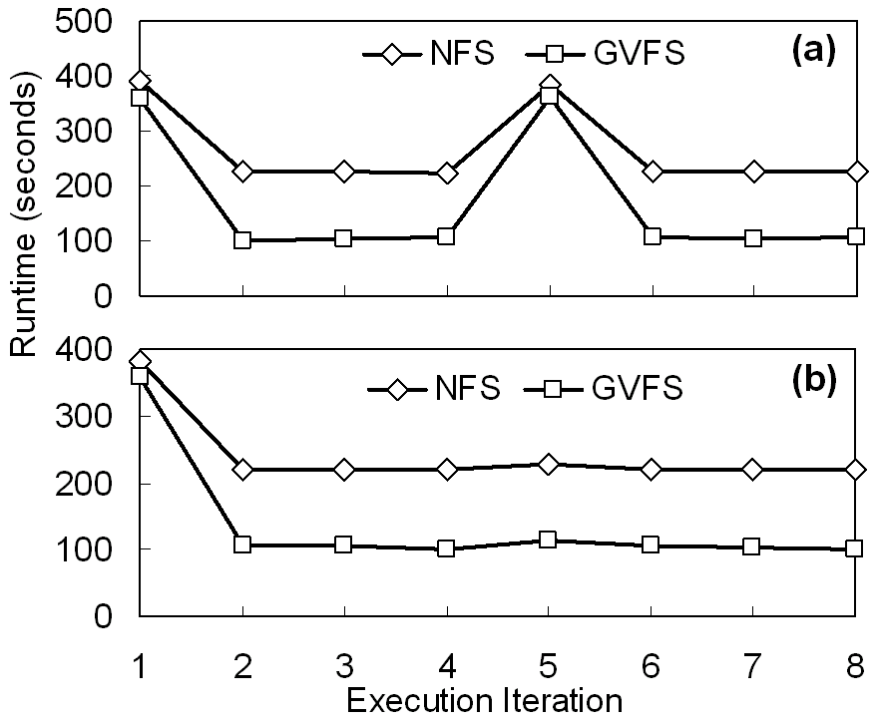

Figure 4-13. Runtimes of the parallel NanoMOS benchmark accessed from a wide-area shared software repository. The benchmark was executed in parallel on six clients, while the repository was maintained by the administrator from another client. The repository was shared among all the clients via native NFS or GVFS with 30s invalidation period. Between the 4th and 5th run an update happened on: (a) the entire MATLAB directory; (b) the MPITB directory only.

among all the clients via native NFS or GVFS with invalidation polling based consistency. The runtimes of the NanoMOS executions are shown in Figure 4-13.

NanoMOS' working data set is relatively small (about 30MB per client), so both NFS and GVFS clients can cache it and reduce the runtime since the second run. But the difference is that the NFS client has to frequently check consistency for the cached data (about 2.7K GETATTRs per client per run), which can be almost eliminated by the GVFS client with its cache consistency. As a result, GVFS delivers more than 2-fold speedup compared to NFS. When an update happens, the NFS client cannot know how many files are affected (the MATLAB package consists of 14K files/directories, but the MPITB has only 540), so it has to always issue the same volume of consistency checks

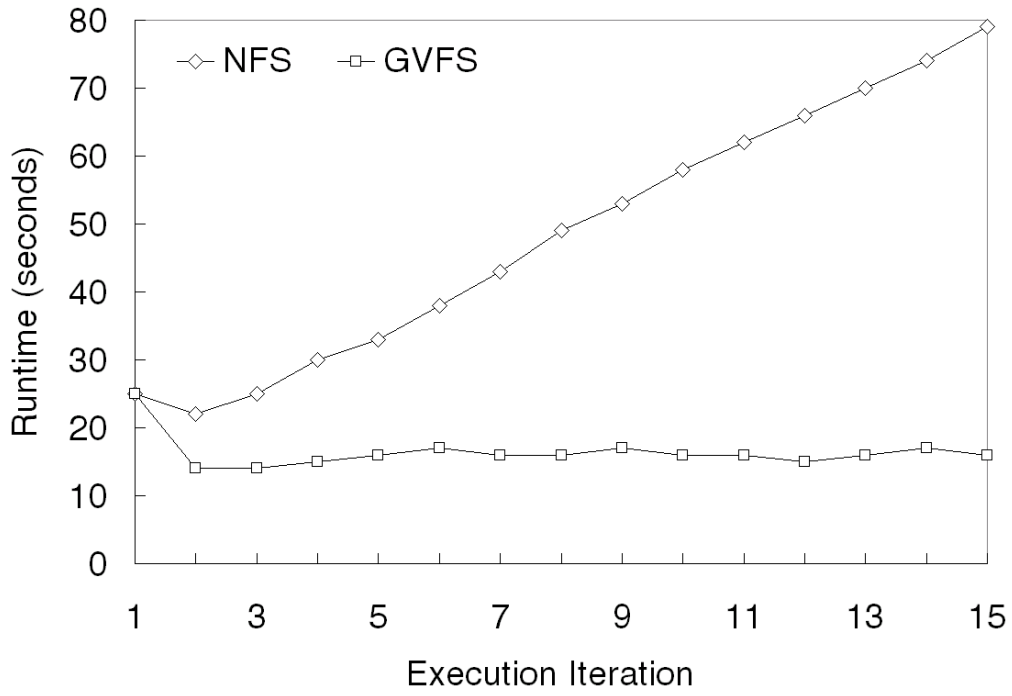

Figure 4-14. Runtimes of consecutive executions of the CH1D data-processing program. Data generation and processing were performed across WAN, where data were shared via native NFS or GVFS with delegation callback based consistency. The data-processing program started each run with 30 more input files from the data-producing program.

for the entire package. However, the GVFS client only uses invalidations proportional to the size of the update and batches them together in a few transactions (MATLAB update needs about 30 GETINV calls per client; MPITB update needs only two calls per client).

4.3.4.6 Scientific data processing

Another benchmark uses a coastal ocean hydrodynamics modeling application, CH1D, to model the distributed scientific data processing scenario discussed in Section 4.1: real-time data are accumulated on coastal observation sites and meanwhile processed on off-site computing centers. A wide-area file system helps the programs to share data naturally without explicitly transferring data back and forth. In the experiment, the data-processing program ran consecutively for 15 times, where each run was started with 30 more input files from the data-producing program. The data were shared between the programs via native NFS or GVFS with the delegation and callback consistency protocol.

Similar to the previous benchmark, the input data set to the data-processing program is small and even 15 runs of data can still fit into its kernel client's memory buffer. However, as the data set grows the amount of consistency that the kernel client has to maintain also increases accordingly. The runtimes of the data-processing program (Figure 4-14) clearly demonstrate this trend: the overhead from maintaining cache consistency increases linearly as the size of the data set. In contrast, with GVFS' consistency protocol this overhead is much smaller and remains practically constant for each run (only 30 callbacks). Accordingly, the performance speedup achieved by GVFS also grows as the data set does and at the 15th run the benchmark runs already 5 times faster than on NFS.

4.4 Security

Security has always been one of the most important concerns for data management in grid-style environments, where data are shared across organizations and domains with limited mutual-trust, and stored and transferred on resources with limited security. However, conventional DFSs designed for the use on LAN support only weak security, because they are often deployed in a relatively more trustworthy environment. For example, NFS (v2 and v3) typically employs UNIX-style authentication with user and group identifiers (IDs), which is difficult to be used across domains. It also does not provide privacy and integrity, and thus NFS RPC messages can be easily spoofed, altered, and forged. Several WAN-oriented DFSs (AFS [25][26], Coda [27]) provide strong security mechanisms, but they need a complex security infrastructure (Kerberos [41]) in place, which requires substantial administrative work from the involved domains.

In addition, conventional DFSs are not designed to support application-tailored, dynamically configurable security mechanisms and policies. Nonetheless, in a grid system, virtual organizations are dynamically established, applications and services are dynamically initiated, and entities and trust are dynamically created. Applications and their execution environments also have very diverse needs for security. In some cases, a cross-domain user and group ID mapping is sufficient for authentication and authorization,

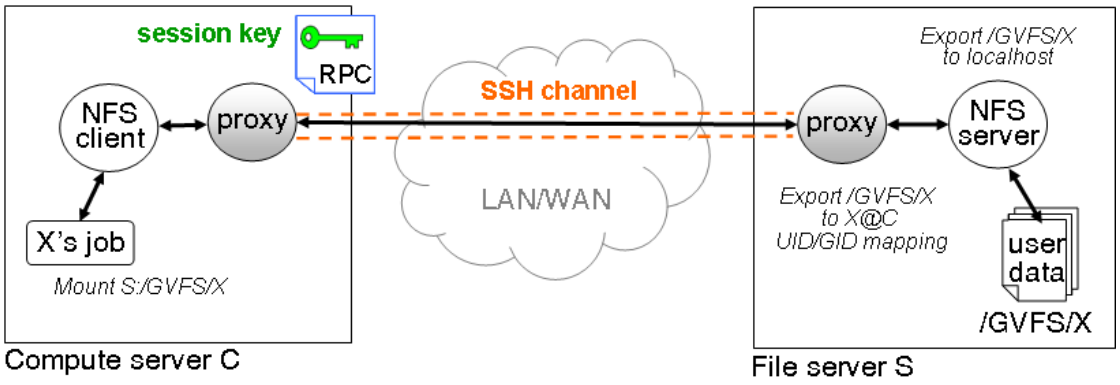

Figure 4-15. Private grid file system built upon GVFS virtualization, SSH tunneling, and session-key based cross-domain authentication.

whereas in others, a security token that uniquely identifies a user across the distributed systems is necessary. To some applications, privacy is not important and thus encryption can be avoided to improve the performance, whereas others may access highly sensitive data and need strong encryption mechanism that requires substantial computation.

Hence, strong security is needed for wide-area DFSs, whereas per-application configuration is also important since it has impacts on both security and performance. GVFS supports both of these goals on top of the virtualization layer by providing strong security mechanisms with high customizability, which are achieved through the two different security approaches discussed in the rest of this section.

4.4.1 Secure Tunneling Based Private Grid File System

In the context of RPC-based applications, security in communication can be provided within or outside the RPC protocol. A key advantage of the latter approach lies in the fact that existing RPC-based clients and servers can be reused without modifications. This subsection describes such an approach taken by GVFS to provide private grid file systems based on secure data tunneling (Figure 4-15).

4.4.1.1 Secure data tunneling

Secure RPC-based connections can be established through the use of TCP/IP tunneling. A tunnel allows the encapsulation and encryption of datagrams at the client side, and corresponding decryption and de-capsulation at the server site. It

supports private communication channels in an application-transparent manner. The application-transparent property of tunneling is a key advantage of this technique and has found wide use in applications such as Virtual Private Networks (VPNs) and secure remote X-Windows sessions.

Tunneling of RPC-based connections can be achieved through mechanisms such as SSL and SSH. The latter is a *de facto* standard for secure logins, providing strong authentication, data privacy, and data integrity for remote login sessions, as well as tunneling of arbitrary TCP connections. GVFS leverages the functionality of SSH to create authenticated, encrypted tunnels between proxy client and proxy server as illustrated in Figure 4-15. Tunneled GVFS connections are TCP-based. This, however, does not prevent GVFS from supporting UDP-based kernel clients and servers. A proxy client can receive RPC calls over UDP from localhost and forward to the proxy server using TCP, and the proxy server can receive RPC calls over the TCP tunnel and forward to localhost using UDP.

The use of SSH to tunnel NFS traffic has been pursued by related efforts, such as Secure NFS [48]. A key differentiator of GVFS from previous approaches is that private GVFS sessions are established *dynamically by middleware on a per-session basis*, rather than statically by a system administrator for groups of users. Another key difference is the GVFS support for per-user identity mappings across network domains.

Per-user tunnels and user mappings are key to establishing dynamic file system sessions in a grid-oriented environment, where users belong to different administrative domains. A secure tunnel multiplexed by users faces the same limitations for cross-domain authentication as NFS, since RPC-based security must be used to authenticate users within a tunnel. With per-user, per file system secure channels, the task of guaranteeing privacy and integrity can be leveraged from the secure channels, whereas the task of authenticating users can be independently carried out by each private file system outside of its channel.

4.4.1.2 Security model

The GVFS private file system channels rely on existing kernel-level services at the client (file system mounting), server (file system exporting), user-level middleware-controlled proxies at both client and server, and SSH tunnels established between them. Hence, the deployment of GVFS involves the setup of appropriate trust relationships between client, server, and middleware.

In the GVFS security model, the data server administrator needs to trust grid middleware to the extent that it allows access to an exported directory tree to be brokered by proxies (e.g., */GVFS/X* on server *S* in Figure 4-15). In essence, the server administrator delegates access control to one or more exported directories to the grid middleware. This is a trust relationship that is similar to those found in other grid data deployments (e.g., GridFTP [6]). The architecture of GVFS allows kernel export definitions to be implemented by local system administrators in a simple manner — the kernel server exports only to the localhost, and exports only the directories that should be accessible via GVFS. Users outside the localhost cannot directly mount file systems from the kernel — only via GVFS proxies. A typical scenario, where a base home directory for grid users is exported through GVFS, requires a single entry in an exports definition file.

Then, the proxies are responsible for authenticating accesses to those file systems exported by GVFS. This is accomplished by means of two mechanisms. The first authentication mechanism is independent from the proxy and consists of the client machine being able to present appropriate credentials (an SSH key or an X.509 certificate for GSI-enabled SSH) to the server machine to establish a tunnel. Second, once the tunnel is established, it is necessary for the proxy server to authenticate requests received through it. Typically, NFS servers authenticate client requests by checking the origin of NFS calls and only allowing those that come from privileged ports of trusted IPs to proceed. In the private GVFS setup, the originator of requests sent to the proxy server is the server's tunnel end-point. Hence, the proxy server receives requests from the localhost

and from non-privileged ports and cannot authenticate the client based on trusted IP/port information. It thus becomes necessary to implement an alternative approach for inter-proxy authentication between tunnel end-points.

The authentication in the private GVFS approach consists of the dynamic creation of a random session key by middleware at the time the proxy server is started and its transmission over a separate private channel to the proxy client (e.g., using GSI-enabled Secure Copy). Then the proxy client appends the key to each NFS procedure call, and the proxy server only authenticates a coming request if it is originated from the localhost and it has a session key that matches the proxy server's key (Figure 4-15). Hence, the use of session keys is completely transparent to kernel clients and servers and requires no changes to their implementations; it only applies to inter-proxy authentication between tunnel end-points. These session keys are used for authentication, similarly to X11/xauth, but not for encryption purposes (the privacy of GVFS is provided by its SSH channel). In the implementation, a session key is a randomly generated 128-bit string and encapsulated in original NFS RPC messages by replacing an unused credential field, so the runtime overhead of supporting this method is very small, consisting of only encapsulation and decapsulation of a session key, and a simple comparison between key values.

The proxy server thus needs to trust the grid middleware to authenticate user credentials and establish an encrypted tunnel, create a random session key, and provide the key to the proxy client through a separate private channel. These mechanisms can be provided by existing grid security infrastructure, such as Globus GSI. Finally, the client administrator needs to trust grid middleware to the extent that it needs to allow NFS mount and unmount operations to be initiated by grid middleware (possibly within a restricted set of allowed base directories, e.g., */GVFS/X* in Figure 4-15). In current GVFS setups, this is implemented with the use of sudo entries for these commands.

4.4.1.3 Evaluation

A prototype of the proposed secure tunneling based private GVFS is evaluated with experiments in this subsection. The experiments were conducted in both local-area and wide-area environments. The file system client is a 1.1GHz Pentium-III cluster node with 1GB of RAM and 18GB of SCSI disk. In the LAN experiment, the file system server is a dual-processor 1.3GHz Pentium-III cluster node with 1GB of RAM and 18GB of disk storage; in the WAN experiment, it is a dual-processor 1GHz Pentium-III cluster node with 1GB RAM and 45GB disk. The LAN setup is a 100Mbps Ethernet and the WAN is through Abilene between Northwestern University and University of Florida. The RTT between the client and server is around 0.17ms in the LAN and around 32ms in the WAN, as measured by RTTometer [97].

The experiments compare the performance of private GVFS (with and without disk caching) with native NFS. In the experiments on GVFS with disk caching, the cache was configured with 8GByte capacity, 512 file banks, and 16-way associativity. The GVFS proxy prototype used here is based on NFSv2, which limits the maximum size of an on-the-wire NFS read or write operation to 8KB. Thus NFSv2 with 8KB block size was used for GVFS. However, in the experiments on native NFS, NFSv3 with 32KB block size was used to provide the best achievable results for comparison. Furthermore, all the experiments were initially set up with cold caches (both kernel buffer cache and possibly enabled proxy disk cache) by unmounting the remote file system and flushing the disk cache if it was used.

The experimental results shown in this subsection consider application-perceived performance measured as elapsed execution time for the following benchmarks:

SPECseis: a benchmark from the SPEC high-performance group. It consists of four phases, where the first phase generates a large trace file on disk and the last phase involves intensive seismic processing computations. The benchmark was tested in sequential mode

with the small data set. It models a scientific application that is both I/O-intensive and compute-intensive.

LaTeX: a benchmark designed to model an interactive document processing session. It is based on the generation of a PDF (Portable Document File) version of a 190-page document edited by LaTeX. It runs the "latex", "bibtex", and "dvipdf" programs in sequence and iterates 20 times, where each time a different version of one of the LaTeX input files is used.

These benchmarks were executed on the clients, where the working directories were either stored on local disks or mounted from the remote LAN or WAN servers. To investigate the overhead incurred by private file system channels, in the LAN environment the performance of native NFS (*LAN/N*) is compared with private GVFS (*LAN/G*) without disk caches. Experiments conducted in the WAN environment must use private GVFS to ensure data privacy and traverse firewalls[2] , but the performance of GVFS without disk caches (*WAN/G*) and with caches (*WAN/GC*) are both compared against the performance of the local disk (*Local*), in order to investigate the overhead of GVFS security and the potential performance improvement achieved by using disk caching.

The experiment results are summarized in Table 4-1. Consider the execution of the LaTeX benchmark. In the LAN scenarios, the overhead of private GVFS (*LAN/N* vs. *LAN/G*) is large at the beginning but is substantially reduced once the kernel buffer cache holds the working data set. This overhead is caused by the latency from both SSH tunneling and proxy processing of RPC calls. The results also show that the kernel buffer cache alone is not sufficient to lower WAN execution time of the LaTeX benchmark in *WAN/G*, but the use of disk caching in GVFS can remarkably reduce the overhead to 17% in *WAN/GC* compared to *Local*. Two factors allow the combination of kernel buffer

[2] Firewalls often restrict network access from outside networks, but SSH connections are typically allowed because of its strong security.

Table 4-1. The overhead of private GVFS for the LaTeX and SPECseis benchmarks. The overhead data are calculated by comparing the execution time of the benchmarks in different scenarios: Local disk (*Local*), LAN on NFS (*LAN/N*), LAN on GVFS (*LAN/G*), WAN on GVFS without disk caching (*WAN/G*) and with disk caching (*WAN/GC*). For the LaTeX benchmark, the comparisons for the execution time of the first iteration, the average execution time of the second to the twentieth iterations, and the total execution time are listed. For the SPECseis benchmark, the comparisons for the execution times of the first phase, the fourth phase, and the total execution time are listed. For both benchmarks, in the *WAN/GC* scenario the write-back cached data were submitted to server after the executions and the needed time was summed into the total execution time.

Overhead	LaTeX			SPECseis		
	1st run	2nd 20th run	total	phase 1	phase 4	total
LAN/G vs. LAN/N	124%	7%	13%	47%	0%	9%
WAN/G vs. Local	797%	180%	21.5%	1500%	1%	265%
WAN/GC vs. Local	691%	17%	60%	24%	0%	26%

and disk caches to outperform a solution with kernel buffer only. First, a larger storage capacity; second, the implementation of a write-back policy that allows write hits to complete without contacting the server.

For the execution of the SPECseis benchmark, the compute-intensive phase (phase 4) as expected achieves very close performance in all scenarios, but the performance of the I/O-intensive phase (phase 1) differentiates very much. It is reasonable to see that the overhead of private GVFS becomes larger as more network communication requires more time for SSH tunneling and proxy processing. This overhead is especially large in *WAN/G* due to the much higher network latency in the wide-area environment; however in *WAN/GC* the use of disk caching effectively hides the latency and significantly reduces the overhead to 24% with respect to *Local*. Besides, the write-back caching also helps to improve performance by avoiding the transfer of temporary data to server. In fact, the benchmark generates hundreds of MBytes of data in the working directory during its calculations, where only tens of MBytes are the required results. In *WAN/GC*, the cached data modifications were written back to the server after the execution and the needed time was summed into its total execution time, which is still less than tenth of that of *WAN/G*.

In overall, the performance of the proposed private GVFS is close to NFS in LAN within 10% for both benchmarks, and with the help from disk caching the overhead in WAN is within 20% for the LaTeX benchmark and within 30% for the SPECseis benchmark relative to the local-disk file system.

4.4.2 The SSL-Enabled Secure Grid File System

The key advantages of the aforementioned secure tunneling based private GVFS are in that existing RPC-based clients and servers can be reused without modifications, and it leverages mature security technologies. However, it requires additional middleware to set up tunnels and keys, and its performance also suffers from the overhead of double user-level forwarding incurred by proxy RPC processing and SSH tunneling. In addition, it is not compatible with widely accepted grid security infrastructure [51], which presents a hurdle to the interoperability with other grid middleware. This subsection presents another security approach that preserve the merits of the secure tunneling based approach and addresses its limitations by protecting RPC communication directly with transport-level security, without the addition of tunneling, and uses widely-accepted grid security tokens to provide compatible authentication and authorization.

4.4.2.1 Design

Secure data access in this approach is provided by transport-level security mechanisms, which enable an efficient secure end-to-end connection between proxy client and proxy server to protect RPC communications. In order to create a secure GVFS session for a grid user to access a file server, public-key based user and server certificates are used to establish the mutual authentication between the proxies. (A user certificate can be the user's grid identity certificate, or a proxy certificate issued by the user that supports delegation [51].) After successful authentication, a shared key is negotiated between the two parties and is used to encrypt the GVFS traffic, whereas the data integrity can also be protected using digital signatures or Message Authentication Code (MAC).

An authenticated user's certificate is used by the proxy server to make authorization decisions, i.e., whether to grant the user's access to the exported files. This is achieved using a grid-style access control mechanism which associates file system access permissions with the grid user's identity embedded in the certificate. Such access control is provided with different granularity which allows for flexible selection based on application needs. For an authorized data access request, the necessary identity mapping is also performed by the proxy server so that the request can be successfully executed on the file server.

The choice of security mechanisms and policies is flexible and customizable per GVFS session, in order to satisfy different security requirements from users and applications. This is important because such configurations have implications on both security and performance. For example, if the data transferred by GVFS are not confidential, encryption can be avoided to improve the data access performance, whereas digital signatures can still be employed to protect its integrity. In contrast, for a GVFS session created for highly sensitive data, encryption must be enabled with strong ciphers which consumes considerable CPU cycles.

4.4.2.2 Implementation

The SSL-enabled secure RPC:

In this approach, secure communication for NFS RPC is achieved using transport-layer security protocols (SSL/TLS [53][54] — referred to generally as SSL in the rest of this subsection). Although secure RPC can be realized at the RPC-layer itself (*RPCSEC_GSS* [44]), several factors have motivated the use of SSL: it has very mature and efficient implementations, which have been successfully employed by many important applications; it supports a wide range of algorithms, which can be leveraged to support flexible security configurations; GVFS sessions are established on per-user/application basis, and thus can use SSL to provide full-featured security without using any RPC-layer mechanisms.

An SSL-enabled secure RPC library has been developed for GVFS based on two key packages, TI-RPC [96] and OpenSSL [55]. TI-RPC (Transport Independent RPC) is the

replacement for the original transport-specific RPC. It allows distributed applications to transparently support RPC over connectionless and connection-oriented transports for both IPv4 and v6. OpenSSL is an excellent implementation of SSL, and it has recently also included the support for datagram protocols (DTLS). Therefore, these tools can be effectively utilized to build a secure RPC library that supports both TCP and UDP.

In this library secure RPC APIs are provided in a way that closely resembles the regular RPC APIs. For example, *clnt_tli_ssl_create* and *svc_tli_ssl_create* are two expert-level APIs for creating a RPC client and server, respectively, using a secure transport for communications. These APIs take the same parameters as their regular counterparts with an additional one for the security configuration structure. The use of authentication, encryption, and MAC as well as their specific algorithms can be specified through this structure and passed on to the library to create secure transports for RPC with the desired security mechanisms.

This secure RPC library is generic to support all RPC-based applications. The fact that both TI-RPC and OpenSSL are stand alone packages helps its use by ordinary users without the need to change any system-level configurations. The current implementation is based on Linux; support for other platforms is also conceivable.

The GSI-based GVFS proxy:

The GVFS proxies are enhanced to use the SSL-enabled secure RPC library for communications, and are also extended with the capability of parsing and validating GSI-based certificates. Using these proxies to establish a grid-wide file system, the privacy and integrity of data access are protected in the secure RPCs, whereas grid authentication and authorization are performed based on the user and server certificates.

A GVFS proxy is configured by a user or service through a configuration file, which is useful for customizing several important aspects of a GVFS session (e.g., the use of disk caching and its parameters) as shown in Figure 4-6. This configuration mechanism is augmented to include the security configurations, including the algorithms

for authentication, encryption, and MAC, and the paths to user, host, and trusted CA certificates. In this way, both the proxy client and proxy server can be properly configured to use the grid user's and server's certificates to authenticate with each other, and set up a data session with the desired security mechanisms and policies.

A GVFS session's security customization can also be reconfigured by signaling the proxies to reload the configuration files. Such dynamic reconfiguration is very useful in several important scenarios. For example, it can force a proxy to reload the certificate when the original one is expired or believed to be breached. It can reset a session's security setup when the desired configuration is changed. It can also be used to force a SSL-renegotiation and refresh the session key for a long-lived session. In fact, a proxy can be configured with a timeout value to enable periodic automatic renegotiation.

Grid file access control:

After successful mutual authentication, the grid user's certificate presented by the proxy client is used by the proxy server for authorization of the data requests received from this session. The user credentials (UNIX user and group ID) in each NFS RPC message are from the client-side account allocated for a grid user or job. They do not represent the grid user's identity and cannot be used for the purpose of authorization, but they are still necessary for cross-domain identity mapping. For each authorized RPC request, these credentials are mapped to a local user account's credentials, which are then used by the kernel NFS server to grant access to files. Such authorization and mapping are determined by the grid file access control policies.

With GSI-enabled proxies, a variety of ACL mechanisms can be employed to enforce access control for GVFS sessions. The basic mechanism is based on a gridmap file which is similar to GSI's gridmap file [51] and provides access control per exported file system. This file describes the mapping between a user's grid identity (distinguished name) in the certificate and a local account's name. If a mapping exists for a user in the gridmap file, the user gains the same access rights to the exported file system as the corresponding local

user. Otherwise, the user is mapped to an anonymous user, or denied of access completely, depending on the session's configuration. In GVFS, the gridmap file can be set up on a per-session basis to enable flexible sharing. For example, if a user wants to share her files with another user, she only needs to add the mapping between that user's distinguished name and her local account name in the session's gridmap file.

Fine-grained access control is realized by leveraging the ACCESS procedure call available in NFSv3 and NFSv4. Each file or directory can have an ACL file associated with it (under the same path and named in the style of *.filename.acl*). A user or service can grant or deny a user's access to a file or directory by putting the user's distinguished name inside the corresponding ACL file along with a bit mask encoding the access permissions. (Only the NFSv3 style ACL is supported in the current implementation.) Upon receiving an ACCESS request, the proxy server checks the user's grid identity against the requested file or directory's ACL, and returns the corresponding bit mask if the user is found in the ACL, or a zero which disables all access permissions.

A file or directory automatically inherits its parent's ACL if it does not have a dedicated ACL file. This inheritance mechanism can reduce the management complexity of ACLs. For the sake of performance, the ACLs are cached in memory by the proxy server once they are read from disk. The ACL files are protected by the proxy server from remote access, and can only be modified by the local owner of the files (typically the GVFS management account, as described below) manually or through an authorized middleware service. Note that in the GVFS security model, the NFS server delegates the access control of the exported file systems completely to the proxies. So the ACL mechanisms in kernel (except for the kernel exports file) are no longer useful for the exported file system and should be disabled to avoid overhead.

4.4.2.3 Deployment

GVFS can be conveniently deployed on grid resources because it does not require any modifications to either applications or kernels. It also obeys the least-privilege principle

in that the proxies and their management services (Section 6.1) work completely at user level and use unprivileged network ports, and they can be managed using a single regular user account (e.g., user *gvfs*) on each host. On the server side, the only privilege required is the configuration of a host-wide exports file used by the kernel NFS server. This can be restricted to a single entry in the exports file by organizing all the grid-accessible file systems under a single path (e.g., */GVFS*), which needs to be exported to only the localhost. On the client side, the use of file system mount and unmount is necessary, and it can also be minimized by giving only the local GVFS management account the permission to use sudo or a setuid program to mount and unmount GVFS sessions to a restricted path (e.g., */GVFS*).

To use GVFS, it is not necessary for a grid user to have a personal account on the client or server. The proxies and their management services create a secure file system session on behalf of the user between the account where her job is running and the account where her files are stored. These job and file accounts are often provided by mapping a grid user to a local user [51], or allocated on-demand for dynamically submitted jobs [87].

4.4.2.4 Evaluation

A prototype of the SSL-enabled GVFS is evaluated in this subsection with experiments. File system benchmarks (IOzone and PostMark) are used to investigate the overhead of achieving strong security under intensive I/O load. Application benchmarks modeling workloads in software development and scientific computing are also employed to study GVFS performance with typical file system usages.

Both LAN and WAN environments were considered in the experiments. LAN-based runs study the overhead from the user-level techniques, whereas tests in an emulated WAN reveal its performance for the target grid environments. NIST Net [95] was used to emulate different wide-area network latencies. The file system client and server as well as the NIST Net router were set up on VMware-based virtual machines. They were hosted on

separated physical servers connected via Gigabit Ethernet. Each physical server has dual 3.2GHz hyperthreaded Xeon processors and 4GB of memory. The client and server VMs both have 1 CPU but with different amount of memory, 256MB and 768MB, respectively.

The use of a network emulator and virtual machines facilitates the quick deployment of a controllable and replicable experimental setup. However, the timekeeping within a virtual machine may be inaccurate so the system clock on a physical server was used to measure time, which suffices the granularity required by this evaluation. All the experiments were conducted on virtual machines running on dedicated physical servers without interference from other workloads. Different (secure) DFS setups were experimented, including:

NFSv3 and **NFSv4**: The native kernel-level NFSv3 and NFSv4 provide the baseline performance for comparison. Although not evaluated here, kernel-level secure NFS solutions (e.g., Kerberos-enabled NFS, GridNFS) can be expected to have worse performance than these results. Kernel NFS implementations use only memory for caching and revalidate the cached data when the file is opened or its attributes have timed out. NFSv4 also provides delegation, which allows a client to aggressively cache data.

GVFS-BASIC and **GVFS-SSH**: The basic GVFS without any security enhancements, and the SSH-enabled private GVFS presented in Section 4.4.1. Their results demonstrate the overhead from the user-level RPC processing and SSH tunneling.

SFS: The related work of Self-certifying File System [30] — another NFS-based user-level secure file system. The overhead from the user-level techniques can also be observed from its performance. SFS aggressively caches attributes and access permissions in memory, which improve the performance for metadata operations.

GVFS-SSL: The proposed SSL-enabled secure GVFS approach. By comparing to the above systems, the experiments examine the performance of the SSL-enabled strong authentication, privacy, and integrity. Aggressive disk caching of attributes, access

permissions, and data were used in the WAN-based tests, so those results also reflect the potential performance improvement from that.

In all of the above setups, the server exported the file system with write delay and synchronous update, and the client accessed the server using TCP and 32KB block size for reads and writes. All the experiment results are reported with the average and standard deviation values from multiple runs. Every run was started with cold client-side caches by unmounting the file system and flushing the disk cache.

IOzone:

The first experiment considers the IOzone [88] benchmark which analyzes a file system's performance by performing read and write operations on a large file with a variety of access patterns. In this experiment, it was executed on the client in read/reread mode, which sequentially reads a 512MB file twice from the server. Since the client has only 256MB of memory, the buffer cache does not help with its LRU-based replacement for the benchmark's sequential reads. In fact, the client needs to read a total of 1GB data from the server during the execution. On the server side, the file is preloaded to the memory before each run, so there is no actual disk I/O involved in the tests. This "extremely" intensive setup reveals the worst-case overhead from GVFS' user-level virtualization and security enhancements.

The experiments evaluate various SSL-enabled GVFS configurations that have different security strengths, as follows:

GVFS-AES uses AES (Rijndael [98]) in CBC mode with 256bit key, a very strong cipher, to encrypt RPC traffic, and ensures data integrity with SHA1-based HMAC [99].

GVFS-RC uses RC4 (ARCFOUR [100]) with 128bit key, a relatively weaker cipher for encryption, and it also enables SHA1-HMAC for data integrity.

GVFS-SHA does not use any encryption but still provides integrity using SHA1-HMAC.

To compare with GVFS, in *GVFS-SSH* the SSH tunnels were configured to use 256bit AES-CBC and SHA1-HMAC, which is similar to the *GVFS-AES* configuration; SFS

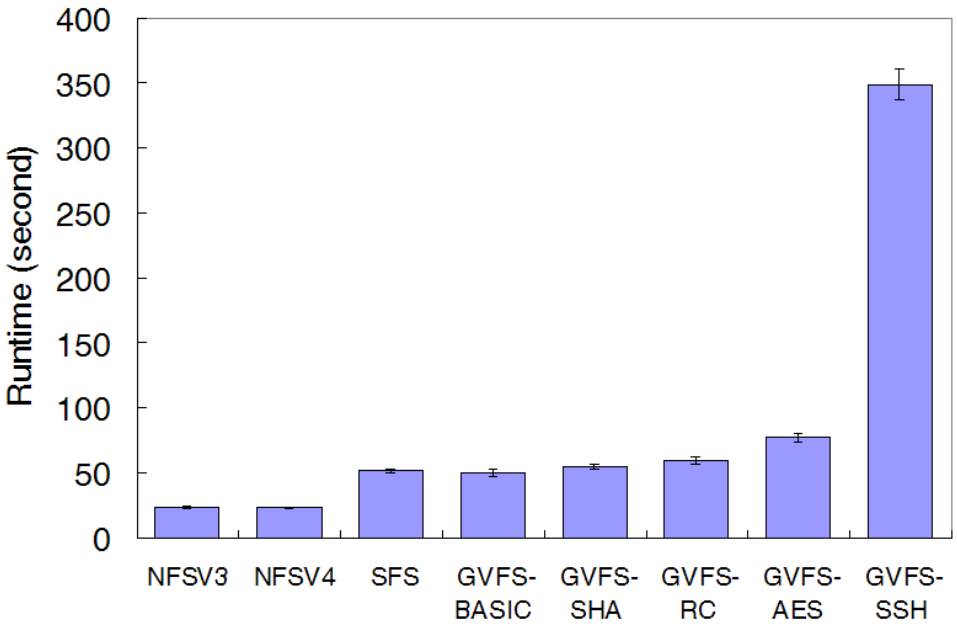

Figure 4-16. Runtimes of IOzone on the different file system setups in LAN: *NFSv3*, *NFSv4*, *SFS*, basic GVFS without security enhancement (*GVFS-BASIC*), secure GVFS with only integrity check but not encryption (*GVFS-SHA*), secure GVFS with RC4 cipher (*GVFS-RC*), secure GVFS with AES cipher (*GVFS-AES*), and SSH-enabled private GVFS (*GVFS-SSH*).

provides privacy and integrity using a customized RC4 and SHA1-HMAC, which is close to the *GVFS-RC* setup.

Figure 4-16 illustrates the runtimes of IOzone on the above DFS setups in LAN. The user-level file systems all show a slow down of more than two-fold compared to the kernel NFS implementations. However, such intensive workload is very rare in practice, and the user-level processing latency can often be overlapped with application "thinking" time or be diminished by disk I/O latency. More importantly, in a WAN environment, the network latency becomes the dominant factor and renders the user-level latency negligible. User-level caching techniques can further hide the latency and improve the file system's performance. These discussions will be validated with the experiments presented later in this subsection.

Comparing the different secure GVFS configurations, *GVFS-SHA* has the lowest overhead from the security enhancements (9% w.r.t. *GVFS-BASIC*), because it only calculates HMAC but does not perform any encryption/decryption on the file system traffic. With the use of encryption, the overhead is increased to 15% in *GVFS-RC*, and 50% in *GVFS-AES*. *GVFS-SSH* has a much higher overhead than the other ones (more than six-fold slowdown w.r.t. *GVFS-BASIC*). This can be at least partially attributed to the penalties from the double user-level forwarding: for every RPC message, two network stack traversals and kernel-user space switches are required by GVFS and SSH to process it. As discussed earlier, such an overhead is highlighted by this intensive experiment setup. The SSL-enabled secure GVFS approach removes this extra penalty, and thus improves the performance substantially[3] .

The experiment also measured the overhead of the user-level file systems in terms of CPU usage. The user time percentages for GVFS proxies and SFS daemons were collected every 5 seconds throughout the benchmark's execution. The client- and server-side results are plotted in Figure 4-17 and Figure 4-18 respectively. On the client, the basic GVFS' CPU usage is very low, averaging 0.6% and under 1% for all the time. For SSL-enabled secure GVFS, the usage goes up to 5% with *SHA1-HMAC*, and further increased to about 8% when encryption/decryption is also used (256bit-AES consumes slightly more CPU than 128bit-RC4). On the server, the CPU usage is even less for *GVFS*, *GVFS-SHA*, and *GVFS-RC*, averaging 0.3%, 1.5%, and 3.6% respectively. All the GVFS configurations need less CPU than SFS which has more than 30% usage on both sides.

PostMark:

[3] The performance of *GVFS-RC* is relative worse than SFS, because the GVFS prototype used in this evaluation is a single-threaded implementation, which cannot handle multiple outstanding RPCs simultaneously. In contrast, SFS makes use of asynchronous RPC and can process several requests at the same time. However, it is reasonable to believe that a multithreaded secure GVFS implementation can deliver much better results, as demonstrated in Section 4.2.2

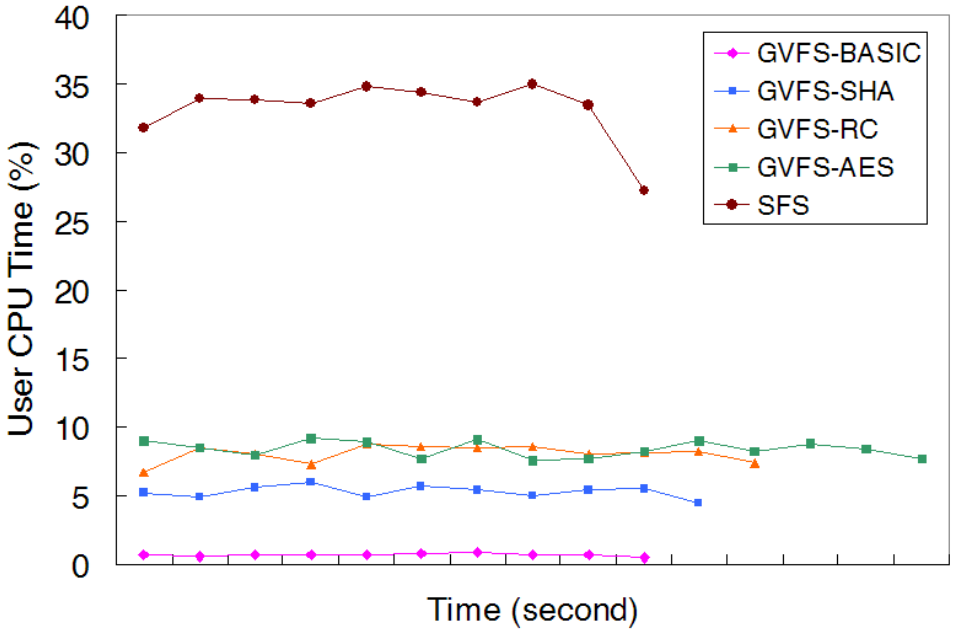

Figure 4-17. Client-side CPU usage of the user-level file system proxy/daemon during the execution of IOzone on different setups: basic GVFS without security enhancement (*GVFS-BASIC*), secure GVFS with only integrity check but not encryption (*GVFS-SHA*), secure GVFS with RC4 cipher (*GVFS-RC*), secure GVFS with AES cipher (*GVFS-AES*), and *SFS*.

The second experiment uses the PostMark [90] benchmark, which is a more realistic file system benchmark that simulates the workloads from emails, news, and web commence applications. It starts with the creation of a pool of directories and files (creation phase), then issues a number of transactions, including create, delete, read, and append, on the initial pool (transaction phase), and finally removes all the directories and files (deletion phase). In contrast to the uniform, sequential data accesses used in the IOzone experiment, the file system is randomly accessed with a variety of data and metadata operations from PostMark.

In this experiment, the initial number of directories and files were 100 and 500 respectively, and the number of transactions was 1000. The transactions were equally distributed between create and delete, and between read and append. The file sizes ranged from 512B to 16KB, and thus the benchmark excised mostly on metadata

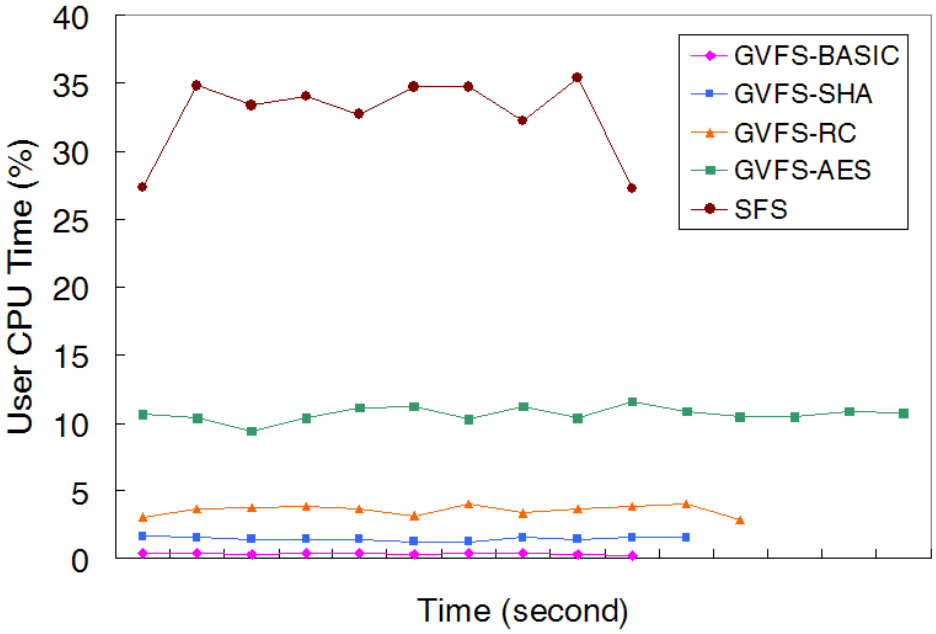

Figure 4-18. Server-side CPU usage of the user-level file system proxy/daemon during the execution of IOzone on different setups: basic GVFS without security enhancement (*GVFS-BASIC*), secure GVFS with only integrity check but not encryption (*GVFS-SHA*), secure GVFS with RC4 cipher (*GVFS-RC*), secure GVFS with AES cipher (*GVFS-AES*), and *SFS*.

operations and small writes. Figure 4-19 shows the runtimes of each PostMark phase for the aforementioned DFS setups. In order to demonstrate the worst-case overhead, the strong GVFS configuration *GVFS-AES* is used for the rest of this subsection and is denoted as *GVFS* or *GVFS-SSL* from here on. For the creation and deletion phases, the runtimes of the secure file systems are all very close to the native NFS' (*GVFS-SSH* is marginally worse than the others). However, for the more intensive transaction phase, where a large number of small data and metadata updates are involved, only GVFS shows a close performance to NFSv3, and it is better than *SFS* and *GVFS-SSH* by 17% and 14% respectively.

The above experiment was conducted in a LAN environment, where the network round-trip time (RTT) between the file system client and server is about 0.3ms. Then it was repeated in the emulated WAN with different network latencies. Figure 4-20 compares

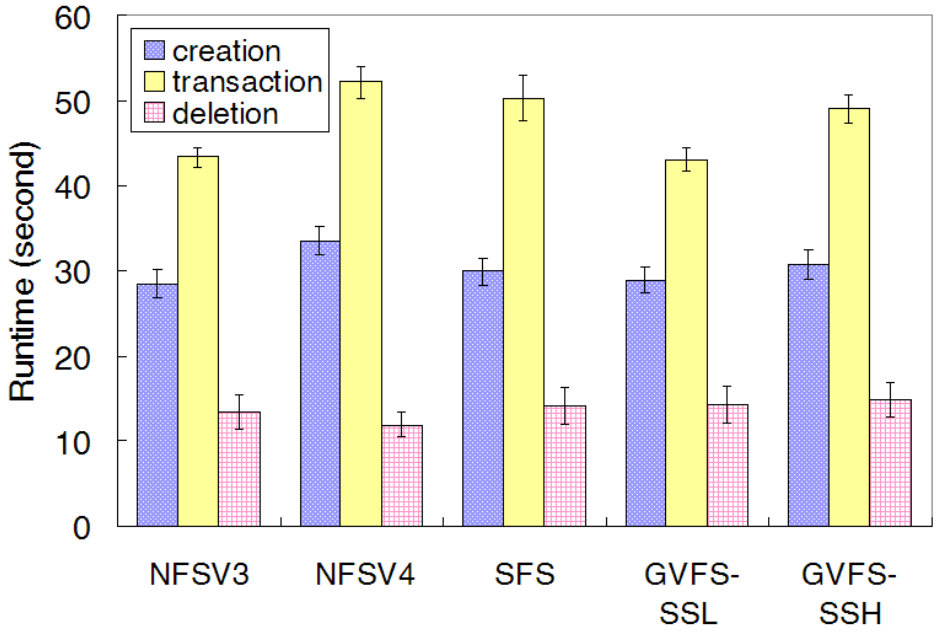

Figure 4-19. Runtimes of various PostMark phases on the different DFS setups in LAN: *NFSv3*, *NFSv4*, *SFS*, SSL-enabled secure GVFS (*GVFS-SSL*), and SSH-enabled private GVFS (*GVFS-SSH*).

the total runtimes of PostMark on *NFSv3* and *GVFS-SSL*. Benefited from the use of disk caching, GVFS shows a very slow decrease in performance as the network latency grows. It is also significantly more efficient than native NFS in wide-area environments, and the speedup is about two-fold when the RTT is 80ms. These results prove the earlier discussions that a user-level secure file system based on GVFS can be very efficient for grid-scale systems.

Since no performance advantage has been observed in the version of NFSv4 used in the experiments, only the results from NFSv3 are reported here as well as in the following experiments, and it is referred to as NFS.

Modified Andrew benchmark:

The third experiment models the typical software development process using a modified Andrew benchmark (MAB). It consists of four phases that exercise different aspects of a file system. The first phase (copy) makes a copy of a software source tree, which transfers a large number of small files within the file system. The second phase

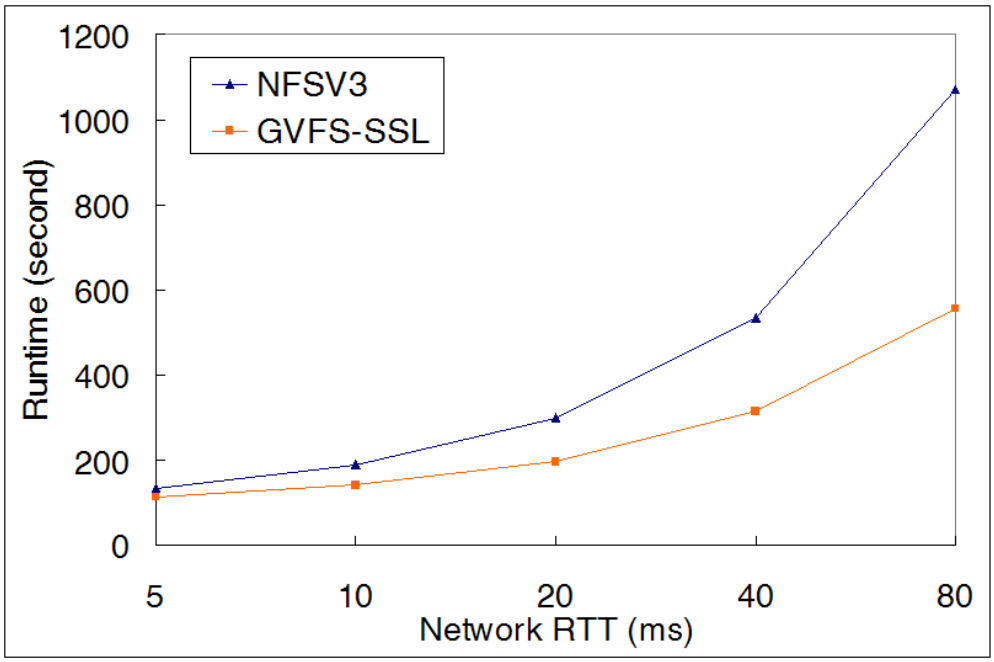

Figure 4-20. Total runtimes of PostMark on NFSv3 and SSL-enabled secure GVFS with varying network round-trip time (RTT).

(stat) recursively examines the status of every file and thus exercises metadata lookups intensively. The third phase (search) reads every file thoroughly to search for a keyword. The last phase (compile) compiles the entire source tree, which generates a large number of data and metadata operations. Because the original Andrew benchmark [25] uses a workload that is too light for today's file systems, the source tree is replaced with the package of an OpenSSH client (openssh-4.6p1). It is a 3-level source tree with 13 directories and 449 files, and the entire compilation generates 194 binaries and object files.

The benchmark was executed on NFS and the proposed SSL-enabled secure GVFS in both LAN and emulated WAN (with 40ms RTT). Their results are shown in Figure 4-21. The secure GVFS performs as well as NFS for the first three phases in LAN, and in the intensive compile phase, it shows a relatively small overhead of 14%. In WAN, GVFS caching effectively hides the network latency, and the total runtime of MAB is slowed down by only 2.5 times. Compared to NFS, it is more than four times faster, and the speedup is approximately nine-fold, five-fold, and eight-fold for the stat, search, and

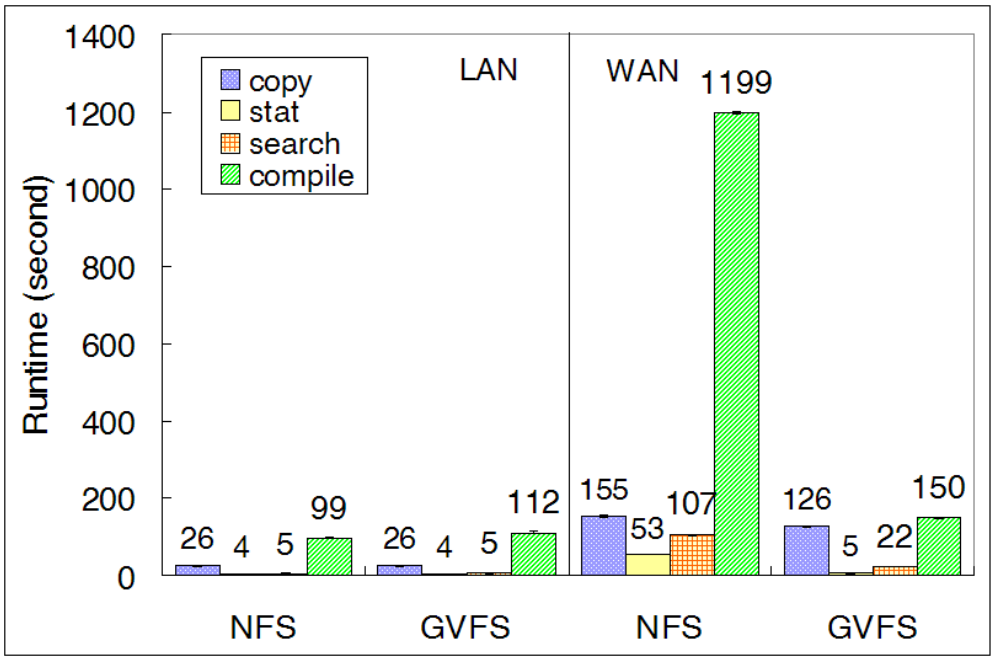

Figure 4-21. Runtimes of various MAB phases on NFS and secure GVFS in both LAN and WAN. The time needed to write back data at the end of execution is 51.2s in average with a standard deviation of 1.3s.

compile phases respectively. Although not shown here, the performance of secure GVFS in LAN can also be improved if disk caching is used, in which the compile phase is only 2% slower than NFS.

SPECseis:

The last benchmark uses a scientific tool, SPECseis, which implements algorithms used by seismologists to locate resources of oil. It is taken from the SPEC HPC96 suite, and its sequential version is considered in the experiment. The execution consists of four phases: (1) data generation, (2) data stacking, (3) time migration, and (4) depth migration. Phase 1 prepares a large initial data file, and each of the following phases performs certain computation based on its previous phase's output file and then generates its own results on disk. In the end, the intermediate outputs are removed and only the results from the last two phases are preserved. This benchmark models a grid application that is both I/O and computation intensive.

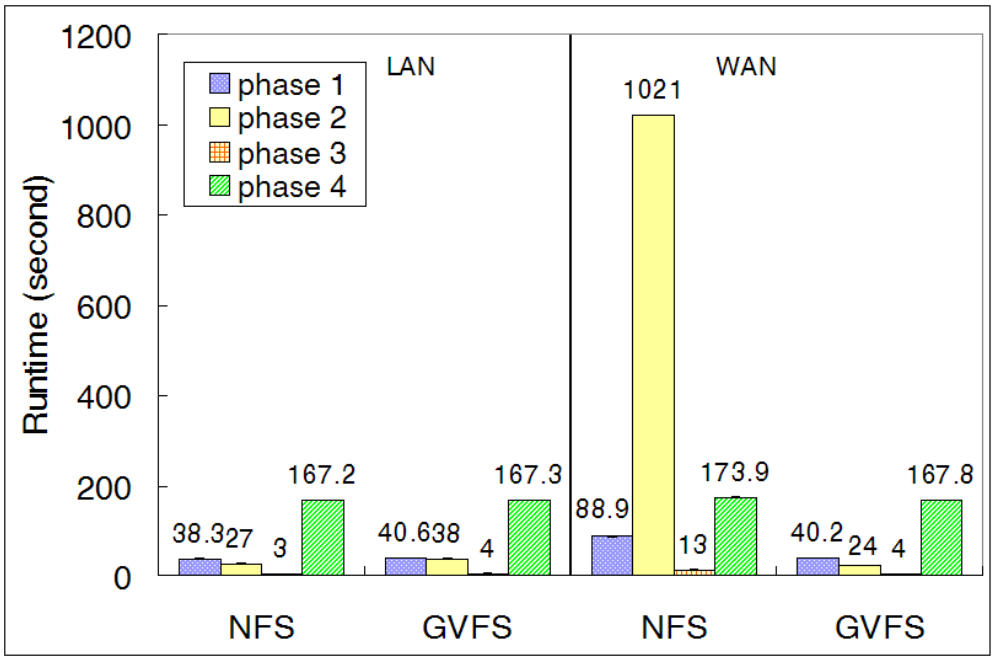

Figure 4-22. Runtimes of various SPECseis phases on NFS and secure GVFS in both LAN and WAN. The time needed to write back data at the end of execution is 14.2s in average with a standard deviation of 1.3s.

This experiment was conducted in both LAN and emulated WAN (with 40ms RTT). Based on the results shown in Figure 4-22, similar observations can be made as in the previous experiment: in LAN, the performance of secure GVFS is very close to NFS; in WAN, GVFS still delivers a good performance and is substantially better than NFS. In phase 1, GVFS stores the large output entirely in cache with the use of write-back; in phase 2, a large number of reads can be satisfied from the data cached in disk, which are not available in memory; and at the end, GVFS also saves considerable time from writing back only the final results but not the temporary data to the server. Consequently, GVFS shows no slow down in WAN. Compared to NFSv3, it is more than five times faster in the total runtime, and the speedup is about two-fold, forty-fold, and four-fold for the first three phases respectively.

4.5 Fault Tolerance

As DFSs getting deployed on WAN, their scale and dynamism also grow dramatically. Unreliable machines and networks often cause the interruption of remote data access

on a DFS and even the loss of data due to unrecoverable failures. In a system built on non-dedicated resources, such as a grid or peer-to-peer system, the dynamic leaving of resources also makes the data stored on them unavailable to others. Therefore, strong tolerance of dynamic failures is critical to using DFSs in such environments, especially to applications that take long time to finish and cannot be easily recovered in case of failures.

Widely deployed DFSs typically do not provide any support for fault tolerance, since they are designed for a relatively reliable and stable environment. However, the GVFS-based virtual DFSs built upon them can employ user-level fault-tolerance protocols to improve the data reliability and availability. In these protocols, fault tolerance is provided by introducing redundancy to the data sets, through the use of replication or coding. Redundancy can be deployed across different servers and different sites to protect against various sorts of failures, including the loss of data due to system crashes, the loss of connections due to network partitions, and the corruption of data due to faulty storage or transmissions. This section presents the replication-based fault-tolerance protocol that makes use of redundancy to improve the reliability of data access on GVFS.

4.5.1 Virtualization of Data Sets

Replication is a common practice for improving reliability. A GVFS-based data session can employ multiple file servers to replicate its data set and provide fault tolerance to the application or user's data access. In such a setup, a proxy server is started on each of the replication servers, allows the connection from the proxy client, and services the client's data requests using its local replica. Data virtualization is provided by the proxy client to hide the different physical location and identity of the replicas, and present a single, consistent view of the data set to the application.

In NFS, a client references files and directories by file handles (FHs), which are typically less than 64 bytes in length and stores contents that are opaque to the client. A file is uniquely identified by its file handle on its file server. However, after replication, a file or directory may have a different FH on every file server, and the client cannot use

the same one to reference its replicas. To address this, virtual FHs are provided by GVFS to virtualize a data set across the replication servers. The proxy client presents virtual FHs to the client and maps them to the physical ones on-the-fly while the data are being accessed.

When a client first mounts a replicated remote file system through GVFS, the proxy client creates a virtual root FH for the virtual DFS, forwards the request to all the servers to obtain the physical root FHs, stores the virtual-to-physical mappings persistently in its local disk caches, and returns the virtual one to the client. Henceforth, the client can use this virtual FH to reference the root in its RPCs, and the proxy client will translate it to the physical root FHs before it forwards the calls to the servers. Similar steps are followed when the client creates a regular file or directory, or looks up one that is not already cached by the proxy.

4.5.2 Replication Schemes

Initial replication of a data set is done by a user or middleware that manages it (please see Section 6.1.3.3 for more details on the replication management). After a GVFS session is created to provide an application with access to the data, the consistency among the replicas is maintained using an active-style protocol [101], with several variations to allow application-tailored customizations. In the basic scheme, every data and metadata update request from the client is multicast to the proxy servers and performed on all the replicas, and it does not return until the proxy client has received an acknowledgement from everyone. Consequently, each replica has the exact same copy of the data set, and if any of them crashes, it has no impact upon the application since the remaining replicas can continue to service as usual.

The above scheme performs updates synchronously and thus limits the performance to the slowest replication server. It can be relaxed to allow asynchronous updates and reduce the overhead for write-intensive applications. In this variation, one of the replication servers, namely the *primary server*, is updated synchronously, whereas the

others are done asynchronously. Specifically, the proxy client responds to an update immediately after the primary server acknowledges it without waiting for the others. (The ordering of requests to a file is provided by the reliable multicast mechanism described below.) Therefore, the performance of the updates can be improved by choosing the fastest server as the primary. It is useful for supporting heterogeneous replication servers, which have, e.g., different level of loads or different network latency to the client.

The synchronization of updates can be further delayed by the proxy client using its disk caches to buffer them locally. Given sufficient cache capacity, the updates are submitted to the servers in batches when the application is idle or completes. In addition to exploiting the locality of updates, the use of write-back reduces the overhead of maintaining replication consistency. The drawback of this scheme is that, in case of client crashes, the updates cannot be recovered until the client is back, or the application need be restarted on another client.

Reliable multicast of updates in the above protocol is realized by client-side logging and multithreaded RPCs. Before the proxy client multicasts a update request, it logs it synchronously on persistent storage. Then it uses multiple threads to forward the update RPCs to the proxy servers concurrently. The RPCs can use both UDP and TCP, and the proxy client will retransmit upon a timeout until a failure is determined. After the request is acknowledged by all the proxy servers, it is considered completed and removed from the log. If the proxy client crashes in the middle of the multicast procedure, then after the recovery, it can check the log and resend the incomplete updates. To enforce the ordering of multicast, during an update multicast, the following requests to the same file will be blocked by the proxy client until the update is completed.

Although it is necessary to propagate an update to the entire replication group, a GVFS session can choose to whether perform read operations on all the replicas or only on the primary one. The former scheme is necessary to provide fault tolerance against Byzantine failures, in which the proxy client compares replies from the servers and returns

one based on the majority vote. This scheme is often combined with the synchronous update approach, and it has a similar disadvantage of limiting the performance to the slowest replication server.

A GVFS session can also use a primary-backup based scheme for read operations, in which the proxy client sends reads only to the primary server. By choosing the server that delivers the best data access performance as the primary, it can not only reduce the overhead from using replication but also improve the data access performance, which is discussed in details in in Section 6.2.3.1. In combination with the above active-style schemes for write operations, this approach in essence provides a "read-one, write-all" fault-tolerance protocol, which can be employed by a read-mostly GVFS session to improve its both performance and reliability.

4.5.3 Application-Transparent Failover

Based on replication and virtualized data sets, fault tolerance can be provided for GVFS data sessions with application-transparent failover. In order to detect a server-side failure happened in a session, the proxy client keeps in touch with the proxy servers through the requests from the application, or by sending periodic heartbeats using NULL RPCs if the application is idle. If a RPC times out, it is first retransmitted in case the failure is caused by a transient network or server problem. When a major timeout (e.g., 100 times of the average response time) is reached before any of the retransmissions succeeds, it assumes that an unrecoverable failure has occurred on that server.

A failed server is excluded from the group of replication servers for the GVFS session, and its application can continue access the remote data using the available servers. If a primary-backup based replication scheme is used and the primary server is failed, the proxy client immediately chooses a new primary server from the backup ones and forwards the failed call as well as the following ones to it. In this way, the failure detection and

recovery is completely masked from the application[4] , and except for a short delay, its data access is not affected at all in this process.

To support timely replica repair and regeneration, a proxy client notifies of a detected failure to the user or the middleware service that manages the GVFS session, so that actions can be taken to recover the replicas either online or offline. GVFS also supports non-interrupt online replica regeneration by allowing a user or service to temporarily switch the GVFS session to the repair mode, in which all the application's updates are buffered by the proxy client in local disk caches and synchronized later with the replication servers after the new replica is generated. The replication management service is discussed in details in Section 6.1.2.

4.5.4 Evaluation

The effectiveness of the replication-based fault-tolerance protocol is evaluated in this subsection with a GVFS data session established for the SPECseis96 benchmark application. The file system client and server were set up on VMware GSX 2.5 based virtual machines (VMs), connected by the WAN between University of Florida (UFL) and Louisiana State University (LSU). The server VM was replicated to provide redundancy for the data set.

During the execution of the benchmark on the client VM, a failure was injected by powering off the server VM. The failure was detected when a timeout of a RPC call happened, and was immediately recovered by establishing a new connection to the replica VM and redirecting the calls through it. The benchmark finished successfully, without being aware of the server failure and recovery during its execution. The elapsed time of such a run (268 seconds) is compared with the execution time of the benchmark in a

[4] To hide a long timeout from the application, the GVFS session needs to be mounted in a "hard" manner, in which the kernel NFS client continues retrying the failed operation indefinitely until it succeeds. If the session is mounted in a "soft" manner, the kernel NFS client will report an I/O error to the application after a long timeout.

normal GVFS session (without injected failure, 258 seconds). The results show that the overhead of error detection and redirection setup is 5 seconds (plus the specified timeout value — 5 seconds, which is adjustable through the proxy). Considering a long-running application, this overhead is negligible.

CHAPTER 5
APPLICATION STUDY: SUPPORTING GRID VIRTUAL MACHINES

The GVFS-based file system virtualization and application-tailored enhancements discussed in the above chapters have been successfully employed to support applications from many different disciplines, including spectroscopy study for biomedical scientists [91] and storm surge modeling for costal researchers [102]. A particularly important and interesting application of this solution is to support virtual machines (VMs) as execution environments for grid applications, where efficient and secure access to both user and VM data is transparently provided to the applications and VMs instantiated on-demand across grids.

5.1 Architecture

A fundamental goal of computational grid systems is to allow flexible, secure sharing of resources distributed across different administrative domains [1]. To realize this vision, a key challenge that must be addressed by grid middleware is the provisioning of execution environments that have flexible, customizable configurations and allow for secure execution of untrusted code from grid users [103]. Such environments can be delivered by architectures that combine system-level VMs [104] and middleware for dynamic instantiation of VM instances on a per-user, per-application basis [17]. Efficient instantiation of VMs across distributed resources requires middleware support for transfer of large VM state files (e.g., memory state, disk state) and thus poses challenges to data management infrastructures.

Mechanisms that present in existing middleware can be utilized to support this functionality by treating VM-based computing sessions as processes to be scheduled (VM monitors) and data to be transferred (VM state). In order to fully exploit the benefits of a VM-based model of grid computing, data management is key: without middleware support for transfer of VM state, computation is tied to the end-resources that have a copy of a user's VM; without support for the transfer of application data, computation is tied to

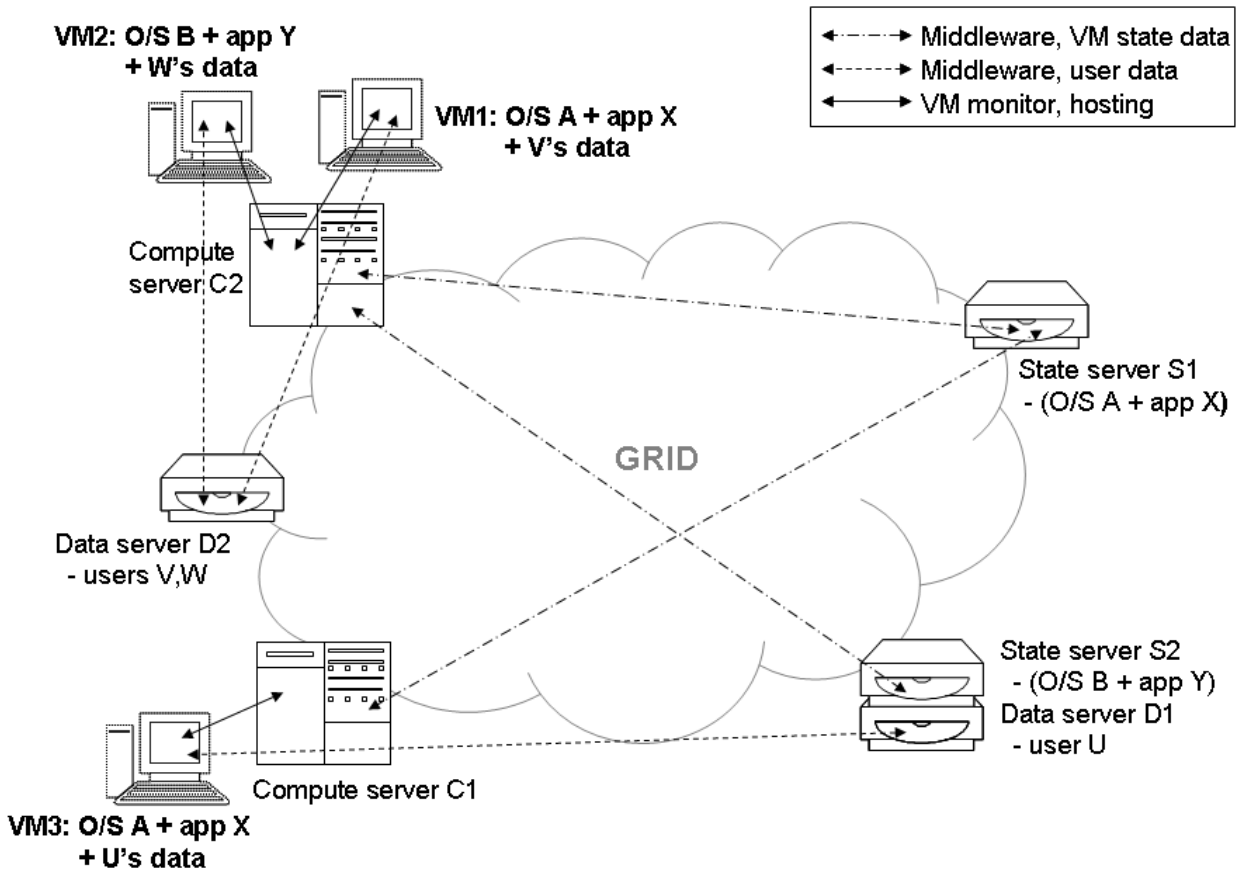

Figure 5-1. The GVFS supported data management for both virtual machine state and user data allows for per-user, per-application VM instantiations (*VM1*, *VM2*, and *VM3*) across grid resources (compute servers *C1* and *C2*, state servers *S1* and *S2*, data servers *D1* and *D2*).

the end-resources that have local access to a user's files. However, with appropriate data management support, the components of a grid VM computing session can be distributed across three different logical entities: the "state server", which stores VM state; the "compute server", which provides the capability of instantiating VMs; and the "data server", which stores user data (Figure 5-1).

As an important application of the GVFS virtualization approach, a data management solution based on GVFS and its application-tailored enhancements is proposed to allow fast dynamic VM instantiation and efficient runtime execution to support VMs as execution environments in grid computing. In particular, as illustrated in Figure 5-2, GVFS disk caching is important to improving the performance of remote VM state access

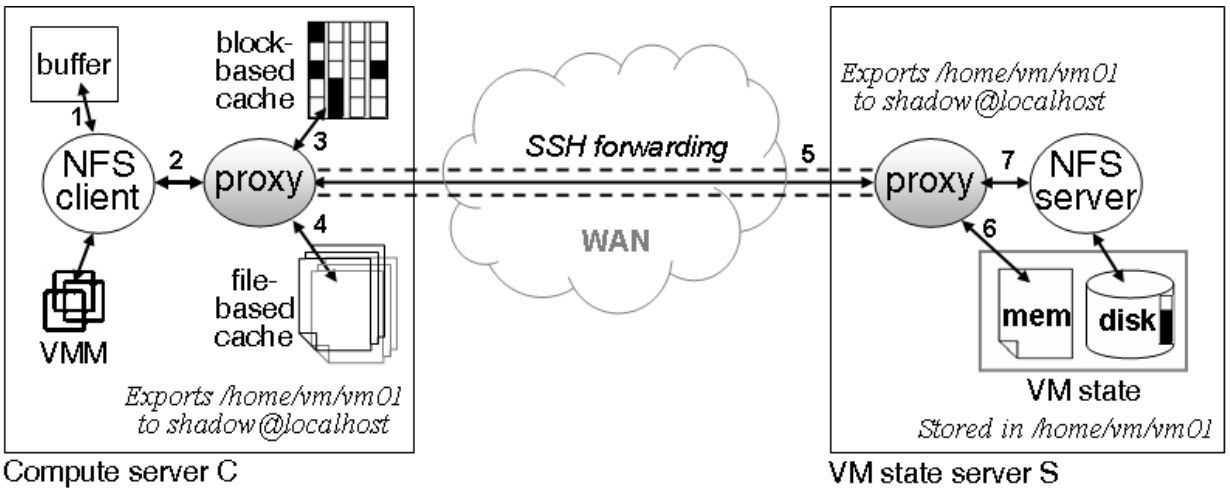

Figure 5-2. The GVFS extensions for VM state transfers. At the compute server, the VM monitor issues system calls that are processed by the kernel NFS client. Requests may hit in the kernel-level memory buffer cache (1); those that miss are processed by the user-level proxy (2). At the proxy, requests that hit in the block-based disk cache (3), or in the file-based disk cache if matching stored metadata (4), are satisfied locally; proxy misses are forwarded as SSH-tunneled RPC calls to a remote proxy (5), which fetches data directly (for VM memory state) (6) or through the kernel NFS server (for VM disk state) (7).

for many VM technologies, including VMware [80], UML [82], and Xen [81], where VM state (e.g., virtual disk, virtual memory) is stored as regular files or file systems. The GVFS private file system channels can also be used to provide secure VM state access over insecure grid resources.

Another user-level extension made to GVFS is the handling of application-specific metadata information. This technique is employed to support grid VMs and realize VM-aware data transfer for on-demand VM instantiation, as discussed in the next section.

5.2 Virtual Machine Aware Data Transfer

The main motivation for metadata handling is to use middleware information to generate metadata for certain categories of files to capture the knowledge of grid applications. Then, a GVFS proxy can take advantage of the metadata to improve data transfer. Metadata contain the data characteristics of the file it is associated with, and define a sequence of actions which should be taken on the file when it is accessed, where

each action can be described as a command or script. When the proxy receives a NFS request to a file which has metadata associated with, it processes the metadata and executes the required actions on the file accordingly. In the current implementation, the metadata file is stored as a hidden file in the same directory as the file it is associated with, and has a special extension (named in the style of *.filename.meta*), so that it can be easily looked up.

For example, resuming a VMware VM requires reading the entire memory state file (typically in hundreds of MBytes or more). Transferring the entire contents of this file over a DFS is time-consuming; however, with application-specific knowledge, it can be pre-processed to generate a metadata file specifying which blocks in the memory state are all zeros. Then, when the memory state file is requested, the proxy client, through processing of the metadata, can service requests to zero-filled blocks locally, ask for only non-zero blocks from the server, reconstruct the entire memory state, and present it to the VM monitor. Normally the memory state contains many zero-filled blocks that can be filtered out by this technique [83], and the traffic on the wire can be greatly reduced while instantiating a VM. For instance, when resuming a 512MB-RAM Red Hat 7.3 VM which was suspended after boot-up, the client issued 65,750 NFS reads, whereas by using this metadata handling technique 92% of the requests could be filtered out without being sent to the server.

Another example of GVFS' metadata handling capability is to help the transfer of large files and enable file-based disk caching. Inherited from the underlying NFS protocol, data transfer in GVFS is on-demand and block-by-block based (typically 4KB to 64KB per block), which allows for partial transfer of files. Many applications can benefit from this property, especially when the working set sizes of the accessed files are considerably smaller than the original sizes of the files. For example, accesses to the VM disk state are typically restricted to a working set that is much smaller (<10%) than the large disk state files. But when large files are indeed completely required by an application (e.g., when a

remotely stored memory state file is requested by VMware to resume a VM), block-based data transfer may become inefficient.

However, if grid middleware can speculate in advance which files will be entirely required based on its knowledge of the application, it can generate metadata for GVFS proxy to expedite the data transfer. The actions described in the metadata can be "compress", "remotely copy", "uncompress", and "read locally", which means when the referred file is accessed by the client, instead of fetching the file block by block from the server, the proxy will: 1) compress the file on the server (e.g., using GZIP); 2) remotely copy the compressed file to the client (e.g., using GSI-enabled SCP [105]) ; 3) uncompress it to the file cache (e.g., using GUNZIP); and 4) generate result for the request from the locally cached file. Once the file is cached all the following requests to the file will also be satisfied locally (Figure 5-2).

Hence, the proxy effectively establishes an on-demand fast file-based data channel, which can also be secure by employing SSH tunneling for data transfer, in addition to the traditional block-based NFS data channel, and a file-based cache which complements the block-based cache in GVFS to form a heterogeneous disk cache. The key to the success of this technique is the proper speculation of an application's behavior. Grid middleware should be able to accumulate knowledge about applications from their past behaviors and make intelligent decisions based on this knowledge. For instance, since for VMware the entire memory state file is always required from the state server before a VM can be resumed on the compute server, and since it is often highly compressible, the above technique can be applied very efficiently to expedite its transfer.

5.3 Integration with VM-Based Grid Computing

The VMs can be deployed in a grid in two different kinds of scenarios, which pose different requirements of data management to the GVFS data sessions created for VM instantiations. In the first scenario, a grid user is allocated a dedicated VM which has a persistent virtual disk on the state server. It is suspended at the current state when the

user leaves and resumed when the user comes back. Nonetheless, the user may or may not start computing sessions from the same server. The VM should be efficiently instantiated on the compute server when the session starts, and the modifications to the VM state from the application execution during the session should also be efficiently reflected on the state server.

The GVFS along with its application-tailored enhancements can well support this scenario in that: 1) the use of metadata handling can quickly restore the VM from its checkpointed state; 2) the on-demand block-based access to the virtual disk can avoid the large overhead incurred from downloading and uploading the entire virtual disk; 3) proxy disk caches can exploit locality of references to the virtual disk and provide high-bandwidth, low-latency accesses to cached file blocks; 4) write-back caching can effectively hide the latencies of write operations perceived by the user/application, which are typically very large in a wide-area environment, and submit the modifications when the user is offline or the session is idle.

In the other scenario, the state server stores a number of template VMs for the purpose of "cloning". These generic VMs have application-tailored hardware and software configurations, and when a VM is requested from a compute server, the state server is searched against the requirements of the desired VM. The best match is returned as the "golden" VM, which is then "cloned" at the compute server [106]. The cloning process entails copying the "golden" VM, restoring it from checkpointed state, and setting up the clone with customized configurations. But instead of copying the entire virtual disk, only symbolic links are made to the disk state files. The golden VM's disk state is read-only shared by all of its clones. After a new clone "comes to life", computing can start in the VM and modifications to the original disk state are stored in the form of redo logs (also known as copy-on-write files). So data management in this scenario requires efficient transfer of the VM state from the state server to the compute server, as well as efficient writes to the redo logs for checkpointing.

Similar to the first scenario, GVFS can quickly instantiate a VM clone by using metadata handling for the memory state file and on-demand block-based access to the disk state files. After the computing starts, the proxy disk cache can help speed up access to the shared virtual disk files, and write-back can help save user time for writes to the redo logs. However, a key difference in this scenario is that a small set of golden VMs can be used to instantiate many clones, e.g., for parallel execution of a high-throughput task. The proxy disk caches can exploit temporal locality among cloned instances and accelerate the cloning process. On the compute server, the cached data of memory and disk state files from previous clones can greatly expedite new clonings from the same golden VMs. In addition, a second-level proxy disk cache can be setup on a LAN server, as explained in Section 4.2.1.3, to further exploit the locality and provide high speed access to the state of golden VMs for clonings to the same LAN.

In both of the above scenarios, middleware-driven cache consistency discussed in Section 4.2.1.2 can be employed. Under a VM management system, such as VMPlant [106] and VMware VirtualCenter [107]: a VM with persistent state can be dedicated to a single user, where aggressive read and write caching with write delay can be used; a VM with non-persistent state can be read-shared among users but each user can have independent redo logs, where read caching for state files and write-back caching for redo logs can be employed. In both cases, the management middleware can control GVFS to write back VM state modifications at the end of the data sessions.

5.4 Evaluation

5.4.1 Setup

This section uses a group of typical benchmarks to evaluate the efficiency of using GVFS to support VM instantiations and executions across network. Experiments were conducted in both local-area and wide-area environments. The LAN setup studies the overhead of using GVFS, whereas the WAN setup investigates its performance in the target grid environments.

The LAN state server is a dual-processor 1.3GHz Pentium-III cluster node with 1GB of RAM and 18GB of disk storage. The WAN state server is a dual-processor 1GHz Pentium-III cluster node with 1GB RAM and 45GB disk. In Section 5.4.2, the compute server is a 1.1GHz Pentium-III cluster node with 1GB of RAM and 18GB of SCSI disk; in Section 5.4.3, the compute servers are cluster nodes which have two 2.4GHz hyper-threaded Xeon processors with 1.5GB RAM and 18GB disk per node. All of the compute servers run VMware GSX server 2.5 to support x86-based VMs. The compute servers are connected with the LAN state server in a 100Mbit/s Ethernet at the University of Florida, and connected with the WAN state server through Abilene between Northwestern University and University of Florida. The RTT from the computer servers to the LAN state server is around 0.17ms, whereas to the WAN state server is around 32ms as measured by RTTometer [97].

In the experiments on GVFS with disk caching, the cache was configured with 8GByte capacity, 512 file banks, and 16-way associativity. The GVFS prototype used here is based on NFSv2, which limits the maximum size of an on-the-wire read or write RPC to 8KB. However, in the experiments on native NFS, NFSv3 with 32KB block size was used to provide the best achievable results for comparison. Furthermore, all the experiments were initially setup with cold caches (both kernel buffer cache and possibly enabled proxy disk cache) by unmounting the remote file system and flushing the proxy cache if it was used. Private file system channels were always employed in GVFS during the experiments.

5.4.2 Performance of Application Executions within VMs

Three benchmarks that represent different typical usage of VMs were used to evaluate the performance of applications executing on GVFS-mounted VM environments:

SPECseis: a benchmark from the SPEC high-performance group. It consists of four phases, where the first phase generates a large trace file on disk and the last phase involves intensive seismic processing computations. The benchmark was tested in sequential mode with the small data set. It models a scientific application that is both I/O-intensive and

compute-intensive. SPECseis is used to study the performance of an application that exhibits a mix of compute-intensive and I/O-intensive phases.

LaTeX: a benchmark designed to model an interactive document processing session. It is based on the generation of a PDF (Portable Document File) version of a 190-page document edited by LaTeX. It runs the "latex", "bibtex", and "dvipdf" programs in sequence and iterates 20 times, where each time a different version of one of the LaTeX input files is used. This benchmark is used to study a scenario where users interact with a VM to customize an execution environment for an application that can then be cloned by other users for execution [106]. In this environment, it is important that interactive sessions for VM setup show good response times to the grid users.

Kernel compilation: a benchmark that represents file system usage in a software development environment, similar to the Andrew benchmark [25]. The kernel is a Linux 2.4.18 with the default configurations in a Red Hat 7.3 Workstation deployment, and the compilation consists of four major steps, "make dep", "make bzImage", "make modules", and "make modules_install", which involve substantial reads and writes on a large number of files.

The execution times of the above benchmarks within a VM, which has 512MB RAM and 2GB disk (in VMware plain disk mode [108]), runs Linux Red Hat 7.3, and stores the benchmark applications and their data sets, were measured in the following four different scenarios:

Local: The VM state was accessed from a local-disk file system.

LAN/G: The VM state was accessed from the LAN state server via GVFS without disk caching.

WAN/G: The VM state was accessed from the WAN state server via GVFS without disk caching.

WAN/GC: The VM state was accessed from the WAN state server via GVFS with disk caching.

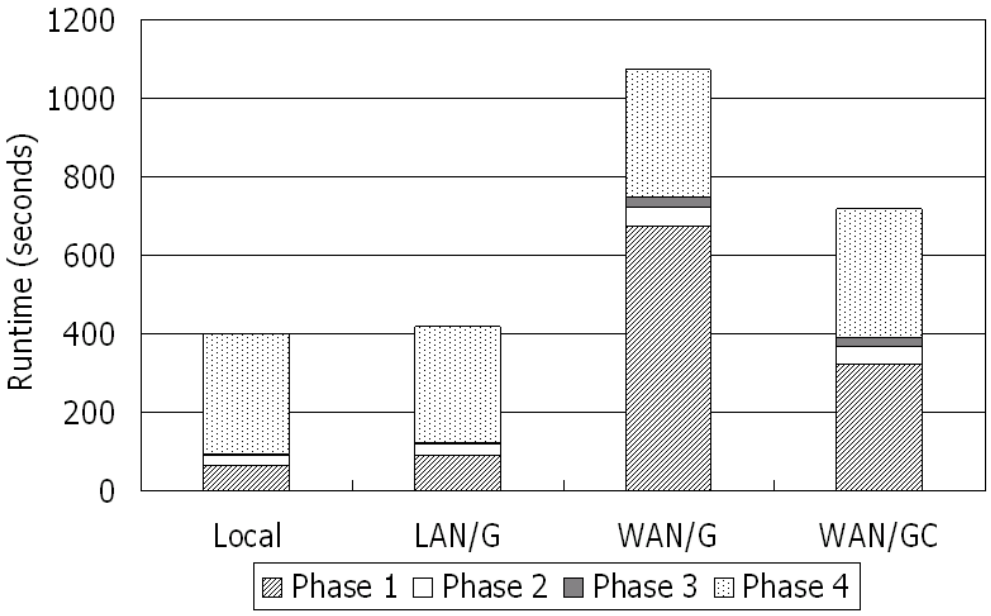

Figure 5-3. The SPECseis benchmark execution time in VM. The results show the runtimes of various SPECseis phases with the VM state accessed from a local-disk file system (*Local*), the LAN state server via GVFS without disk caching (*LAN/G*), the WAN state server via GVFS without disk caching (*WAN/G*), and the WAN state server via GVFS with disk caching (*WAN/GC*).

Figure 5-3 shows the execution times of the four SPECseis phases. The performance of the compute-intensive part (phase 4) is within a 10% range across all scenarios. The results of the I/O-intensive part (phase 1), however, shows a large difference between the *WAN/G* and *WAN/GC* scenarios — the latter is faster by a factor of 2.1. The benefit of a write-back policy is evident in the phase 1, where a large file that is used as an input to the following phases is created. The use of disk caching in GVFS also brings down the total execution time by 33 percent in the wide-area environment.

The LaTeX benchmark results in Figure 5-4 show that in WAN interactive users would experience a startup latency of 225.67 seconds (*WAN/G*) or 217.33 seconds (*WAN/GC*). This overhead is substantial when compared to *Local* and *LAN*, which execute the first iteration in about 12 seconds. Nonetheless, the start-up overhead in these scenarios is much smaller than what one would experience if the entire VM state has to be downloaded from the state server for data access (2818 seconds). During subsequent

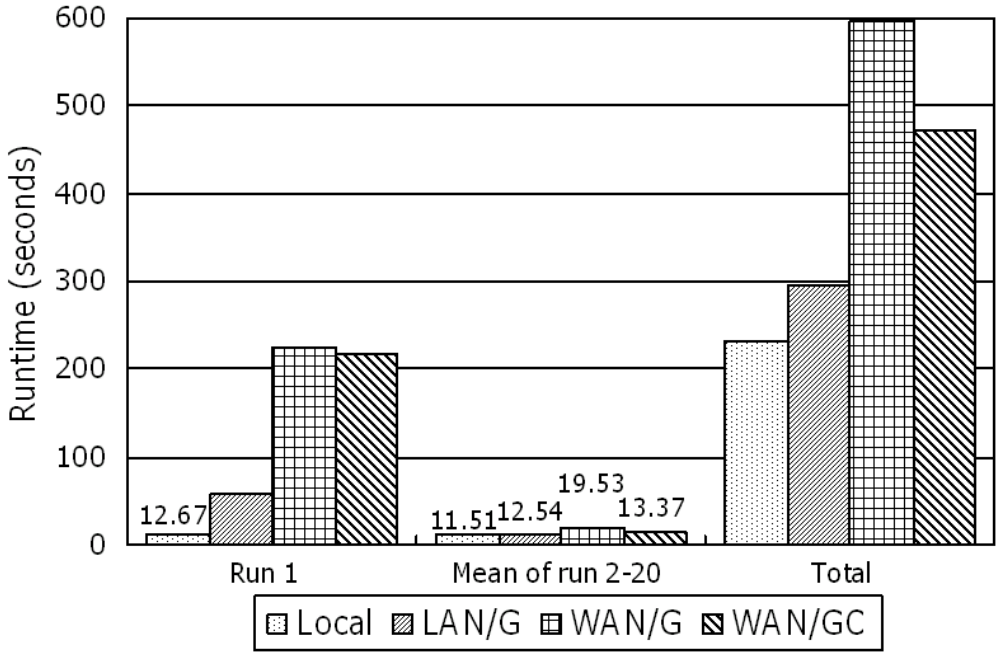

Figure 5-4. The LaTeX benchmark execution time in VM. The results show the runtime of the first run, the average runtime of the following 19 runs, and the total runtime, with the VM state accessed from a local-disk file system (*Local*), the LAN state server via GVFS without disk caching (*LAN/G*), the WAN state server via GVFS without disk caching (*WAN/G*), and the WAN state server via GVFS with disk caching (*WAN/GC*).

iterations, the kernel buffer can help to reduce the average response time for *WAN/G* to about 20 seconds. The use of GVFS disk caching can further improve it for *WAN/GC* to very close to *Local* (8% slower) and *LAN/G* (6% slower), which are 54% faster than *WAN/G*. The time needed to submit cached state modifications is around 160 seconds, which is also much shorter than the uploading time (4633 seconds) of the entire VM state in the download-upload data access model.

Experimental results from the kernel compilation benchmark are illustrated in Figure 5-5. The first run of the benchmark in the *WAN/GC* scenario which begins with "cold" caches shows an 84% overhead compared to that of the *Local* scenario. However, for the second run, the "warm" caches help to bring the overhead down to 9%, and compared to the second run of the *LAN* scenario, it is less than 4% slower. The availability of disk caching allows *WAN/GC* to outperform *WAN/G* by more than 30 percent. As

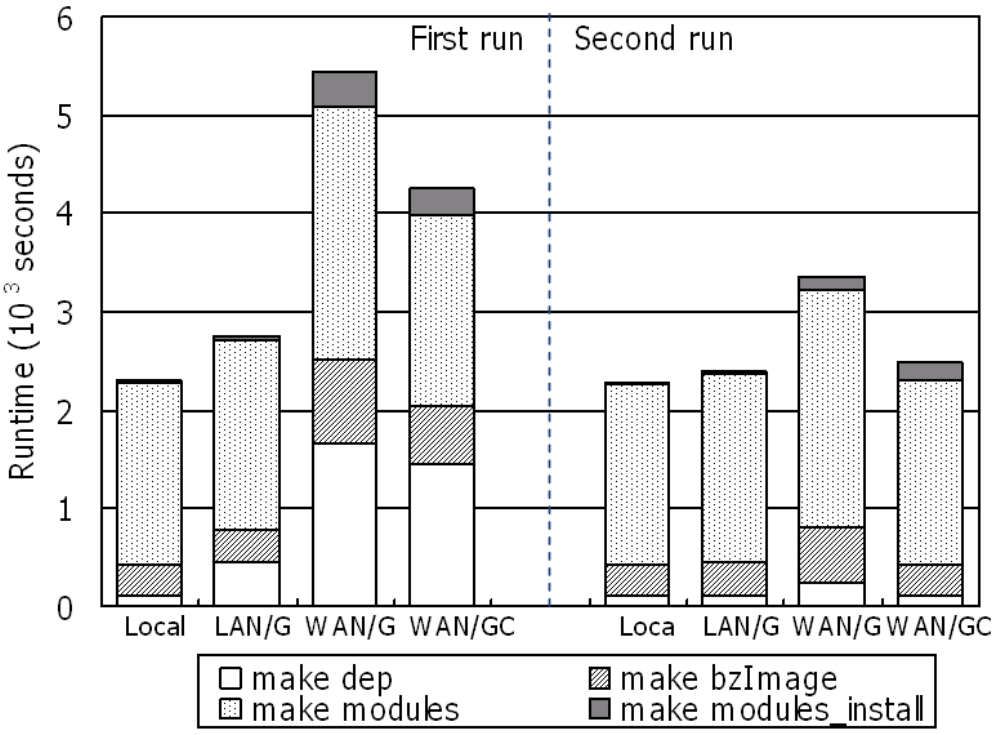

Figure 5-5. Kernel compilation benchmark execution time in VM. The results show the runtime for four different phases in two consecutive runs of the benchmark, with the VM state accessed from a local-disk file system (*Local*), the LAN state server via GVFS without disk caching (*LAN/G*), the WAN state server via GVFS without disk caching (*WAN/G*), and the WAN state server via GVFS with disk caching (*WAN/GC*).

in the LaTeX case, the data show that the overhead experienced in an environment where program binaries and/or data sets are partially reused across iterations (e.g., in application development environments), the response times of WAN-mounted GVFS sessions are acceptable.

5.4.3 Performance of VM Cloning

Another benchmark is designed to investigate the performance of VM cloning under GVFS. The cloning scheme is as discussed in Section 5.3, which includes copying the configuration file, copying the memory state file, building symbolic links to the disk state files, configuring the clone, and at last resuming the new VM. The execution time of the benchmark was also measured in five different scenarios:

Local: The VM was cloned for eight times sequentially from a local-disk file system.

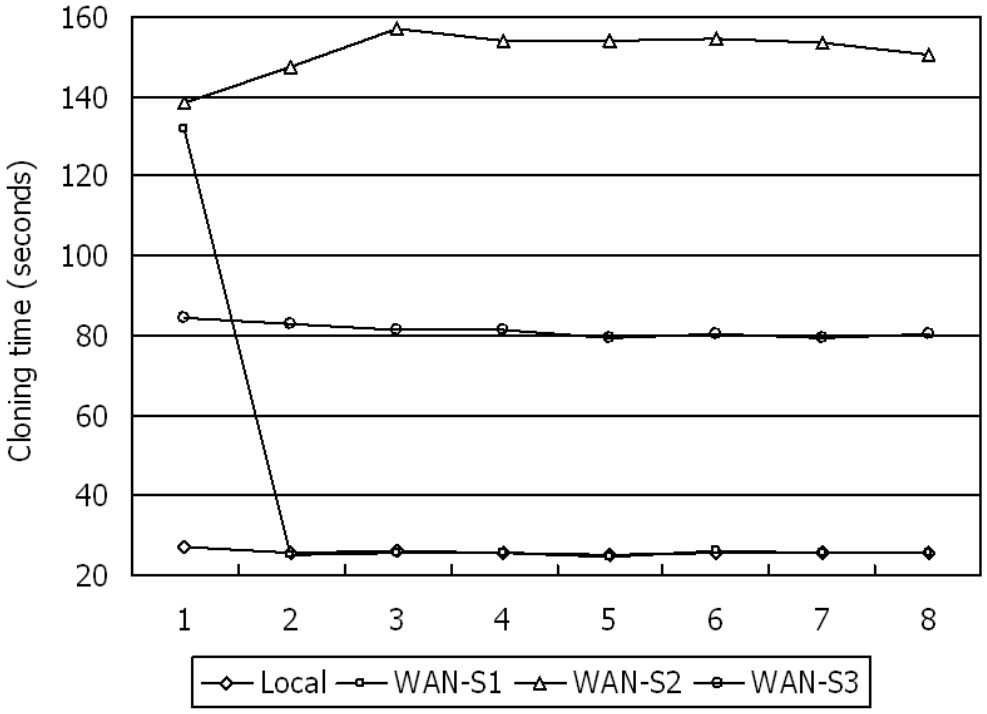

Figure 5-6. Performance of a sequence of VM clonings (from 1 to 8). Each cloned VM has 320MB of virtual memory and 1.6GB of virtual disk. The results show the VM cloning time in the different scenarios.

WAN-S1: The VM was cloned for eight times sequentially from the WAN state server via GVFS. The clonings were supported by GVFS with private file system channel, proxy disk caching, and metadata handling. It is designed to evaluate the performance when there is temporal locality among clonings.

WAN-S2: The setup is the same as WAN-S1 except that eight different VMs were each cloned once to the computer server sequentially. It is designed to evaluate the performance when there is no locality among clonings.

WAN-S3: The setup is the same as WAN-S2 except that a LAN server was used to provide a second-level proxy disk cache to the compute server. Eight different VMs were cloned, which were new to the compute server, but were pre-cached on the LAN server due to previous clones to other computer servers in the same LAN. This setup is designed to model a scenario where there is temporal locality among the VMs cloned to the same LAN.

Table 5-1. Total times of cloning eight VMs in WAN-S1 and WAN-P when the caches (kernel buffer cache, proxy block-based cache, and proxy file-based cache) are cold and warm.

	Total time when caches are cold	Total time when caches are warm
WAN-S1	1056 seconds	200 seconds
WAN-P	150.3 seconds	32 seconds

WAN-P: Eight VMs were cloned in parallel from the WAN state server to eight compute servers via GVFS.

Figure 5-6 shows the cloning time for a sequence of VMs which have 320MB of virtual memory and 1.6GB of virtual disk. In comparison with the range of GVFS-based cloning times shown in the figure, if the VM is cloned using Secure Copy for full-file copying, it takes approximately twenty minutes to transfer the entire state. If the VM state is not copied but is read from a native NFS-mounted directory, the cloning takes more than half an hour because the block-based transfer of memory state file is slow. However, the enhanced GVFS with proxy disk caches and metadata support to compress (using GZIP) and transfer (using SCP) the VM's memory state can greatly speed up the cloning process to within 160 seconds. Furthermore, if there is temporal locality of accesses to the memory and disk state files among the clones, GVFS even allows the cloning to be performed within 25 seconds if data are cached on local disks or within 80 seconds if data are cached on a LAN server.

Table 5-1 compares the performance of sequential cloning with parallel cloning via GVFS. In the experiment of *WAN-P*, the eight compute servers shared a single state server and GVFS proxy server. But when the eight clonings started in parallel, each proxy client spawned a file-based data channel to fetch the memory state file on demand. The speedup from parallel cloning versus sequential cloning is more than 700% when the caches are cold and more than 600% when the caches are warm. Compared with the average time to clone a single VM in the sequential case (*WAN-S1*), the total time for cloning eight VMs in parallel is only 14% longer with cold caches and 24% longer with

warm caches, which implies GVFS' support for VM cloning can scale to parallel cloning of a large number of VMs very well. In both scenarios, the support from GVFS is on demand and transparent to user and VM monitor. In addition, as demonstrated in Section 5.4.2, following a VM's instantiation via cloning, GVFS can also improve its run-time performance substantially.

CHAPTER 6
SERVICE-ORIENTED AUTONOMIC DATA MANAGEMENT

The data management system proposed in this dissertation is built upon two levels of software. The first level is based on the GVFS approach described in the previous chapters, which addresses the problem of providing transparent and application-tailored data access in grid-style environments. Based on GVFS, data sessions can be created on demand for applications upon the shared physical resources as illustrated in Figure 3-1. These sessions can be independently customized to provide the desired data access according to their application needs, e.g., to serve the diverse applications described in Section 4.1.

The second level of the proposed data management system addresses the problem of managing data provisioning in a large, dynamic system, i.e., how to manage many dynamic and diverse GVFS data sessions in a large-scale system. It is desirable that data management can leverage the knowledge of applications (characteristics, usage scenarios, service quality requirements) to optimize the data provisioning, and that it can be done automatically according to high-level objectives (e.g., Quality of Service). This chapter describes service-oriented middleware designed towards these goals. Service-based management hides the complexity of data provisioning from clients — users or other middleware, and transparently prepares data for their applications through interoperable interfaces. Intelligence can be further embedded into the services to automatically optimize data access according to the QoS goals specified by clients.

6.1 Service-Based Data Management

6.1.1 Architecture

Figure 6-1 illustrates the overall architecture of the proposed service-oriented data management. It supports dynamic management of grid-scale data provisioning by means of service-based management middleware (the *control flow*, dashed lines), and GVFS-based

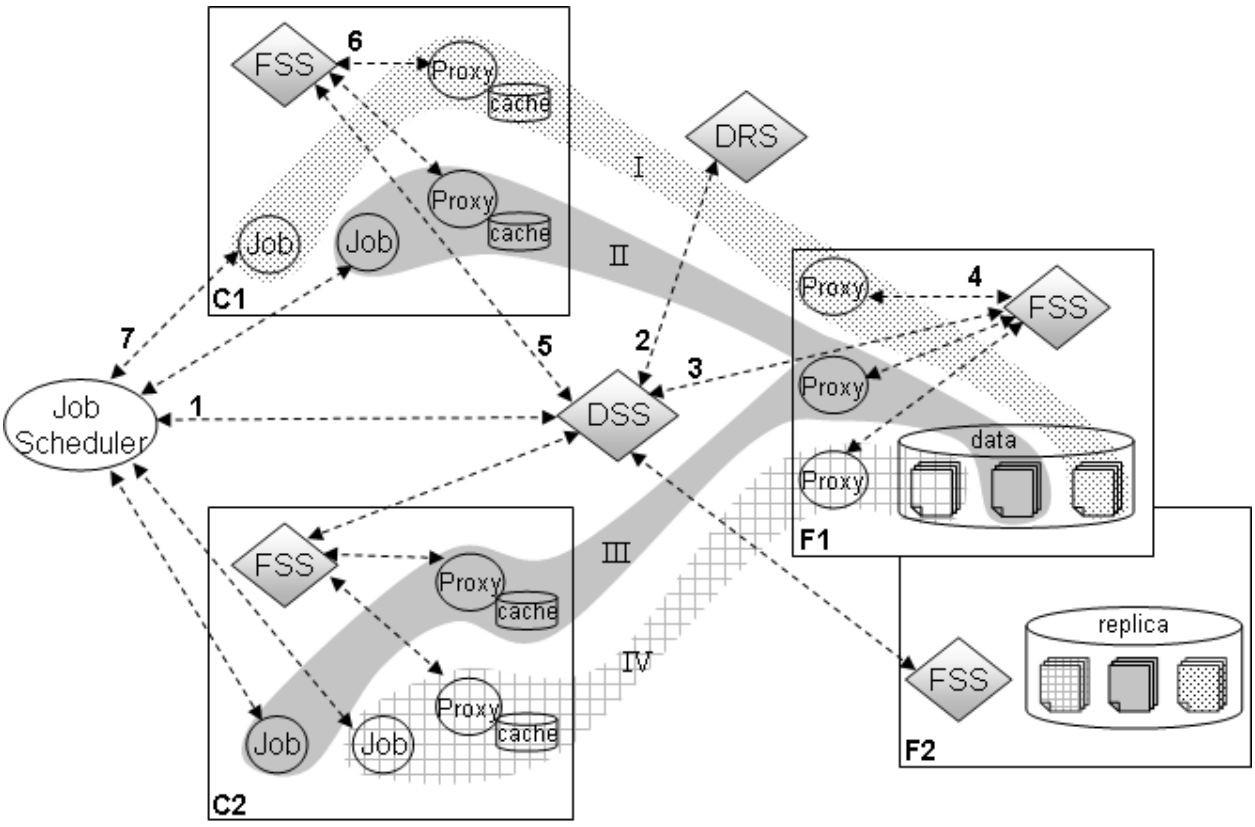

Figure 6-1. Example of GVFS sessions established by the data management services on compute servers (*C1*, *C2*) and file servers (*F1*, *F2*). In step 1, the job scheduler requests the DSS (Data Scheduler Service) to start a session between *C1* and *F1*; step 2, the DSS queries the DRS (Data Replication Service) for replica information; it then requests in step 3 the FSS (File System Service) on *F1* to start the proxy server (step 4). The DSS also requests the FSS on *C1* to start the proxy client and mount the file system (steps 5, 6). The job scheduler can then start a task in *C1* (step 7), which can access the data on server *F1* through session I. Sessions II, III and IV are isolated from session I.

data sessions[1] (the *data flow*, shaded regions). The figure shows examples of data sessions established by the data management services. Sessions are independently configured and isolated from each other through the services and proxies. Several sessions can also share the same data set by connecting to the same proxy server (e.g., Session II and III in Figure 6-1).

[1] The service also supports file-based data transfers for the data flow, as described in Section 6.1.3.1.

Fundamentally, the goal of this architecture is to enable flexible, secure resource sharing. This involves the establishment of relationships between providers and users that are complex (and often conflicting) in distributed environments. From a user's standpoint, resources should ideally be customizable to their needs, regardless of their location. From a provider's standpoint, resources should ideally be configured in a single, consistent way. Otherwise, sharing is hindered by a provider's inability to accommodate individual user needs (and associated security risks) and by the user's inability to effectively use systems over which they have limited control.

To this end, the proposed service-oriented approach builds upon two key aspects of the Web Service Resource Framework (WSRF): *interoperability* in the definition, publishing, discovery, and interactions of services [109][68][110], and *state management* for controlling data access sessions that persist throughout the execution of an application. It also builds upon a virtualized data access layer that supports user-level customization. As a result, the services are deployed once by the provider, and can then be accessed by authorized users to create and customize independent data access sessions.

These data management services are intended for use by both end-users and middleware brokers (e.g., job schedulers) acting on their behalf. In either case, it is assumed that the user or middleware client can authenticate to the service host, directly or indirectly through delegation, leveraging authentication support at the WSRF layer, and obtain access to a local user identity on the host (e.g., via GSI-based grid-to-local account mappings, or via middleware-allocated, "logical" user accounts [111][112]). (See Section 6.1.4 for more details about the security architecture.)

The remaining of this chapter is organized as follows. Section 6.1.2 presents the details of the proposed data management services. Section 6.1.3 describes how they are employed to manage application-tailored data sessions. Section 6.1.5 discusses several usage examples.

6.1.2 The WSRF-Based Data Management Services

6.1.2.1 File system service

The File System Service (FSS) runs on every compute and file server and controls the local GVFS proxies. It implements the establishment and customization of file system sessions through the management of the proxies. The proxy processes are the stateful resources to the service, and the service provides the interface to start, configure, monitor, and terminate them. Their state (e.g., proxy configurations) is stored in files on local disk. A proxy client is associated with a single session; a proxy server, however, can be involved in more than one session to support the data sharing among multiple applications (e.g., session II and III in Figure 6-1).

The service customizes a proxy via configurations defined in a file and can signal it to dynamically reconfigure itself by reloading the file (Figure 4-6). The configuration file holds information including disk cache parameters, cache consistency protocol, security configuration, and data replica location. They are represented as WS-Resource Property and can be viewed and modified with standard WSRF operations (*getResourceProperty* and *setResourceProperty*). When the FSS receives a request for a session's status, it signals the proxy to report the accumulated statistics (number of RPC calls, resource usage etc.) and to issue an NFS NULL call to the server to check whether the connection is still alive.

6.1.2.2 Data scheduler service

The Data Scheduler Service (DSS) is in charge of creation and management of GVFS sessions. These sessions are associated to the service as its stateful resources and their state (e.g., session configurations) is maintained in a database. The service supports the operations of creating, configuring, monitoring, and tearing down of a session.

A request to create a session needs to specify the two endpoint locations (client's IP address and mount point, server's IP address and file system export path) and the desired configurations of the session in the aspects of caching, multithreading, consistency,

security, and reliability. The DSS first checks its information about other sessions to resolve sharing conflicts. For example, if the same data set is accessed by another session with write-back caching enabled, the service can interact with the corresponding FSS to force the session to submit the cached data modifications and disable write-back afterwards.

When there is no conflict, the DSS can proceed to start the session (Figure 6-1). It asks the server-side FSS to start the proxy server and the client-side FSS to start the proxy client and then establishes the connection. Before sending a request to the client-side FSS, the DSS also queries the DRS (a service described below). If there are replicas for the data set, their locations are also sent along with the request, so that in case of failure the session can be redirected to a backup server.

Note that a session is set up for a particular job submitted by a user or service. If there is an irresolvable data sharing conflict when scheduling a session (e.g., the data set is currently under exclusive access by another session), the DSS cannot establish the session and it returns an error to the service client.

The DSS can also dynamically reconfigure a session's configurations and monitor its status as needed, via the configuring and monitoring operations on the session's proxies through the corresponding FSSs. It associates the resource ID of the session with the resource IDs of the session's proxies, so that it can reference the concerned proxies during its interactions with the FSS.

6.1.2.3 Data replication service

The Data Replication Service (DRS) is responsible for managing data replication, and its stateful resources are the data replicas. The service exposes interfaces for creating, destroying, and querying a given data set's replicas. The state of the replicas is stored in a relational database, which facilitates the query and manipulation of information about replicas. The service can be queried with the location (IP address of the server and

path to the data on the server) of a data set (primary or backup one), and it returns the locations of all the replicas.

A request to create a replica needs to specify the locations of the source data and the desired replica. If a replica does not already exist at the requested location, the DRS then interacts with the DSS to schedule a session between the source and destination and have the data replicated. Such a data session can employ efficient bulk-data transfer mechanisms described below (Section 6.1.3.1) to achieve high-throughput replication, especially for wide-area data replication. A replica can also be destroyed as needed through the DRS, which contacts the DSS to schedule the replica removal from the specified location, after it becomes unused by existing sessions. Whenever a replica is created or destroyed, the DRS updates the database accordingly.

The prototype of the above described data management services has been built using WSRF::Lite [71], a Perl-based WSRF implementation of WSRF. The database needed to store the resource state for DSS and DRS has been implemented using MySQL.

6.1.3 Application-Tailored Data Sessions

The data management services are capable of creating and managing dynamic GVFS sessions. Unlike traditional distributed file systems which are statically set up for general-purpose data access, each GVFS session is established for a particular task. Hence the services can apply application-tailored customizations on these sessions to enhance data access in the aspects of performance, consistency, security, and fault tolerance. Figure 6-2 illustrates the use of several such enhancements on a session to improve data access performance and reliability. The following three subsections describe the choices that can currently be made on a per-application basis.

6.1.3.1 Grid data access and file transfer

The FTP-based tools can often achieve high throughput for large-size file movements [6], but the application's data access pattern needs to be well defined to employ such utilities. For applications which have complex data access patterns and for those

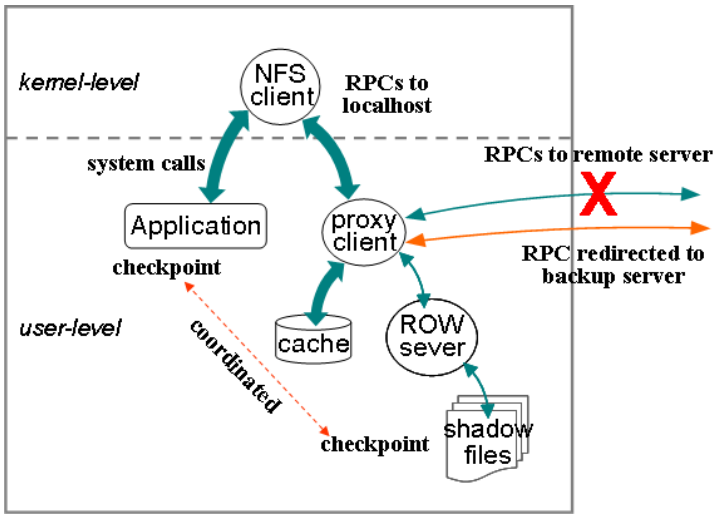

Figure 6-2. Application-tailored enhancements for a GVFS session. Read requests are satisfied from the remote server or the proxy cache. Writes are forwarded to the loopback ROW server and stored in shadow files. When a request to the remote server fails it is redirected to the backup server.

that operate on sparse data sets, the generic file system interface and partial-data transfer supported by GVFS are advantageous. Both models are supported by the data management services.

The FSS can configure data access sessions based on file system proxies. According to the information about the job accounts and file accounts provided by the DSS, the FSS dynamically sets up cross-domain identity mappings (e.g., remote-to-local user/group IDs) on a per-session basis. The FSS can configure the GVFS session with application-tailored caching, security, and reliability configurations as discussed later. It is also capable of dynamically reconfiguring a GVFS session based on changed data access requirements, for example, when a session's data set becomes shared by multiple sessions.

The services can also employ high-throughput data transfer mechanisms (e.g., GridFTP [6], SFTP/GSI-OpenSSH [105]) if it is known in advance that applications use whole-file transfers. This scenario can be dealt with in two different ways. In the conventional way, a user or service authenticates through the DSS, which requests the FSS to transfer files on behalf of the user: downloading the required inputs and presenting

them to the application before the execution; uploading the specified outputs to the server after the execution.

The FTP-style data transfer can also be exploited by GVFS *while maintaining the generic file system interface*. The proxy client uses this functionality to fetch the entirely needed large files to a local cache, but the application still operates on the files through the kernel NFS client and proxy client in a block-based fashion. In this way, the selection of data transfer mechanism becomes transparent to applications and can be leveraged by unmodified applications. Such an application-selective data transfer session has been shown to improve the performance of instantiating grid virtual machines (VMs) [20] in Section 5.2, and it can be applied to support other applications as well through the data management services.

Block-based or file-based disk caching can be employed by data sessions to support the above different data transfer mechanisms, leveraging locality to improve remote data access performance (6-2). Each session's cache can be independently customized in terms of both parameters (size, associativity) and policies (read-only, write-through, write-back), according to its application's characteristics and requirements.

6.1.3.2 Cache consistency

Applications can also benefit from the availability of different cache consistency models. The DFS and FSS services enable applications to select well-suited strong or weak consistency models by dynamically customizing GVFS sessions. Different cache consistency protocols are overlaid upon the native NFS client-polling mechanism by the user-level proxies as discussed in Section 4.3.

Typical NFS clients use per-file and per-directory timers to determine when to poll a server. This can lead to unnecessary traffic if files do not change often and timers are set to too small a value on one hand, and long delays in updates if timers have large values on the other hand. Across wide-area networks, revalidation calls contribute to

long application-perceived latencies. In contrast, the overlaid models customize the use of consistency checks on a per GVFS session basis.

Because the data management services dynamically establish sessions that can be independently configured, the overlaid consistency protocol can be selected to improve performance when it is applicable. Two examples where overlaid consistency protocols can improve performance are described below:

Single-client sessions: when a task is known to the scheduler to be independent (e.g., in high-throughput task farm jobs), aggressive client-side caching can be enabled for both read and write data and completely satisfy consistency checks locally to achieve the best possible performance. As writes are delayed on the client, the data may become inconsistent with the server. But from the session's point of view, its data can be guaranteed to be consistent by the DSS. Consistency actions that apply to a session are initiated through the DSS in two occasions: 1) when the task finishes and the session is to be terminated, the cached data modifications are automatically submitted to the server; 2) when the data are to be shared with other sessions, the DSS reconfigures the session by forcing it to write back cached modifications and disable write-back henceforth. In either case, the DSS waits for the write-back to complete before it starts another session on the same data.

Multiple-client sessions: For GVFS sessions where exclusive write access to data is not necessary or not possible, the scheduler can apply relaxed cache consistency protocols on these sessions to improve performance, e.g., the invalidation-polling based consistency discussed in Section 4.3.2. For applications that cannot tolerate any inconsistency, strong consistency protocols can be employed by their sessions, such as the delegation-callback based consistency introduced in Section 4.3.3.

6.1.3.3 Fault tolerance

Reliable execution is crucial for many applications, especially long-running computing and simulation tasks. The data management services currently provide two techniques for

improved fault tolerance: client-side ROW-assisted checkpointing, and server replication and session redirection (Figure 6-2).

Redirect-on-Write (ROW) file system [113]: The services can enable ROW on a GVFS session, so all file system modifications produced by the client are transparently buffered in local stable storage. In such a session, the proxy client splits the data requests across two servers: reads go to the remote main server, and writes are redirected to a local ROW server[2] . This approach relies on the opaque nature of NFS file handles to allow for virtual handles that are always returned to the client, but mapped to different physical file handles at the main and ROW servers. Files whose contents are modified by the client have "shadow" files created by the ROW server in a sparse file, and block-based modifications are inserted in-place in the shadow file.

When an application is checkpointed, the FSS can request the checkpointing of all buffered modifications in the shadow file system. Then, when the recovery from a client-side failure is needed, as the application is rolled back to the previous saved state, the FSS also rolls back the application's data modifications to the corresponding state. Without the ROW mechanism, when the application rolls back the modifications on the files since the last checkpointing are already reflected on the server, in which case the data state becomes inconsistent with the application state and the recovery cannot proceed correctly. For instance, files already deleted on the server may be needed by the application again after it is rolled back, which will cause the application to fail. A number of checkpointing techniques can be employed in this approach, including [114][115]. One particular case is the checkpointing of an entire VM which includes both the application instance and the ROW file system. This is demonstrated in the following experiment.

[2] Reads of files that have been modified by the client are routed to the ROW server, instead of the main server.

This experiment models a scenario where a VM running an arbitrary application is checkpointed, continues to execute, and later fails. Before failing, the application changes the state of the file server irreversibly — e.g., by deleting some temporary files. This case was tested with the Gaussian computational chemistry application running on a compute VM and using data mounted from a data VM. The experiment shows that, in native NFS, when the compute VM was resumed to its previous checkpointed state, the temporary files needed by the application were already gone on the data VM, so NFS reported a stale file handle error and the application aborted. In contrast, with GVFS and the ROW enhancement, the data state was stored along with the application state as part of the VM's checkpoint, and the file deletions were buffered locally and did not take place on the data VM. So when the compute VM was resumed from the checkpoint, the application was recovered successfully, because the data modifications happened after the checkpoint were discarded from the ROW and the data on the data VM were still in a consistent state with the application.

Server replication and session redirection: Replication is a common practice for fault tolerance, and it is employed in GVFS to improve the reliability of data sessions as discussed in Section 4.5. The data management services support the use of server replication and session redirection to tolerate server-side failures, including server crashes and network partitions, as follows. To prepare a GVFS session with replicated data sets, the DSS checks the DRS for information about the data set's existing replicas (the IP address of a replica server and the path to the replica on the server). If the data set does not have enough number of replicas to support the application-needed replication degree (the number of replicas for the data set), the services can be asked to create the replicas on the desired file servers.

The DSS then requests the FSS on each replica server to start a proxy server for this session. It also passes on the replica information to the FSS on the client, so that the proxy client can be started with connections to all the proxy servers. During such

a session, the proxy client virtualizes the data set across the replicas and transparently detect a failure as well as recover from it based on the fault-tolerance protocols described in Section 4.5. The choice of replication degree, replica placement, and consistency scheme for a GVFS session can all be customized by the services according to the application's requirements.

The data management services can also customize the security configurations of GVFS data sessions in terms of the security policies and mechanisms discussed in Section 4.4. However, a complete security architecture is needed to provide security to not only the data sessions but also the management services, and these two levels of security need to be consistent with each other and be compatible with other security middleware. The following subsection presents such a security architecture designed to achieve these goals.

6.1.4 Security Architecture

This dissertation proposes a two-level security architecture for GVFS-based grid data management (Figure 6-3). It leverages transport-level security to protect the file system traffic of GVFS and employs message-level security to secure the interactions with the management services. Both levels utilize widely-accepted security tokens (X.509 [43]/GSI certificates [51]) to support grid user authentication and file access control. The design and implementation of the file system level security are already discussed in Section 4.4.2. Therefore, the rest of this subsection focuses on the service-level security and its interaction with the file system level security.

To create a GVFS session through the services, a grid user or a service that acts on behalf of the user needs to authenticate with DSS using the user's certificate. Authorization is performed by checking the certificate against an access control list (ACL), or consulting a dedicated authorization service. An authorized user can then proceed to delegate the management services the right to create a GVFS session on behalf the user: the DSS uses the user's certificate to interact with the client- and server-side

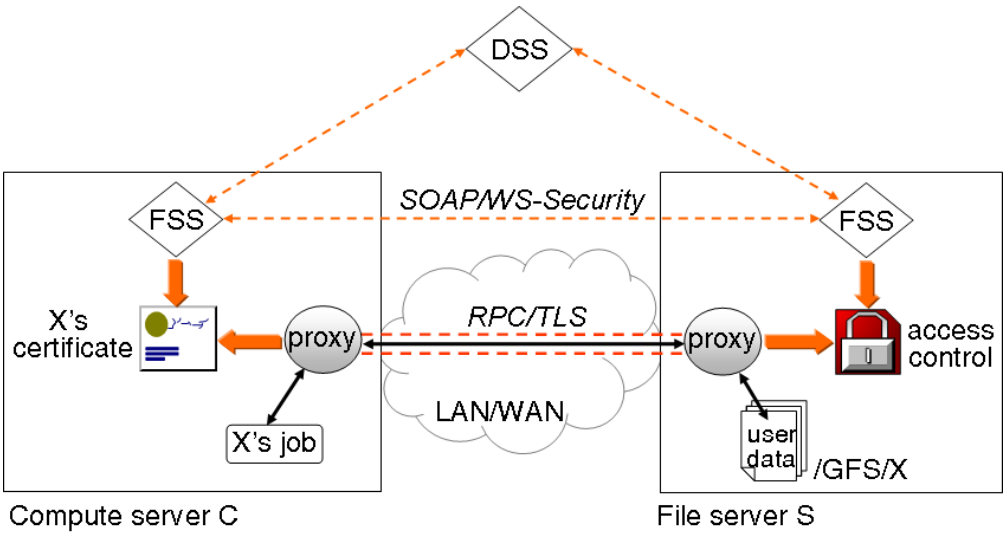

Figure 6-3. Security architecture of GVFS-based data management system. Transport-level security is leveraged to protect the data access on GVFS, while message-level security is employed to secure the interactions with the management services. Grid user certificates and ACLs are used for authentication and access control.

FSSs, which in turn configure the proxies to use this certificate to establish a secure file system session.

Security for service interactions is enabled at the message level (Figure 6-3). Despite the performance inefficiency, message-level security offers great functionality at the service level, which is necessary for the management services to use and interact with other high-level services. These services are not in the critical path of grid data access; they are only involved infrequently when control is needed on a GVFS session, specifically, when creating, configuring, and destroying a session. Therefore, the use of more expensive security mechanisms does not hurt a GVFS session's I/O performance and has negligible overhead compared to a session's lifecycle.

The data management services are implemented using WSRF::Lite [71], a Perl-based implementation of WSRF (Web Services Resource Framework [68]). This tool supports signing and verifying of SOAP messages using X.509 certificates according to the WS-Security standard [56], which is utilized to enable grid security at the service level.

The resulting data management services can securely communicate with each other, use the grid user and server certificates to perform authentication and authorization, and then control the GVFS proxies to use these certificates for the file system level security.

As mentioned earlier, the management services are responsible for creating and customizing GVFS-based data sessions on behalf of grid users or services. These operations are provided through Web service interfaces, which conform to the WSRF standard; meanwhile, the security of the service-level interactions also follows Web service security standards and it is compatible to GSI. Such compatibility is important to providing interoperability with other grid services, e.g., to serve a job scheduler which needs to prepare data access for the jobs submitted to a grid resource.

The management services support flexible grid file access control for GVFS sessions using the mechanisms discussed in Section 4.4.2.2. Per file system ACLs are stored in the DSS database and are used to automatically generate the sessions' gridmap files. For fine-grained access control, the services also provide an interface to manage the per-file/directory ACLs stored along with the exported files and directories. However, in a large grid system, the access control to grid resources is often dedicated to the virtual organization's security service (e.g., a Community Authorization Service [116]). The GVFS services can consult such a special security service for file access control decisions at the granularity of individual grid users or groups of users.

6.1.5 Usage Examples

This subsection describes two examples of using the management services discussed above to manage GVFS data sessions for two important applications.

6.1.5.1 Virtual machine based grid computing

As discussed in Chapter 5, GVFS has been applied to support VM based grid computing. VMs have been demonstrated as an effective way to provide secure and flexible application execution environments in grids [17]. The dynamic instantiation of VMs requires efficient data management: both VM state and user/application data need

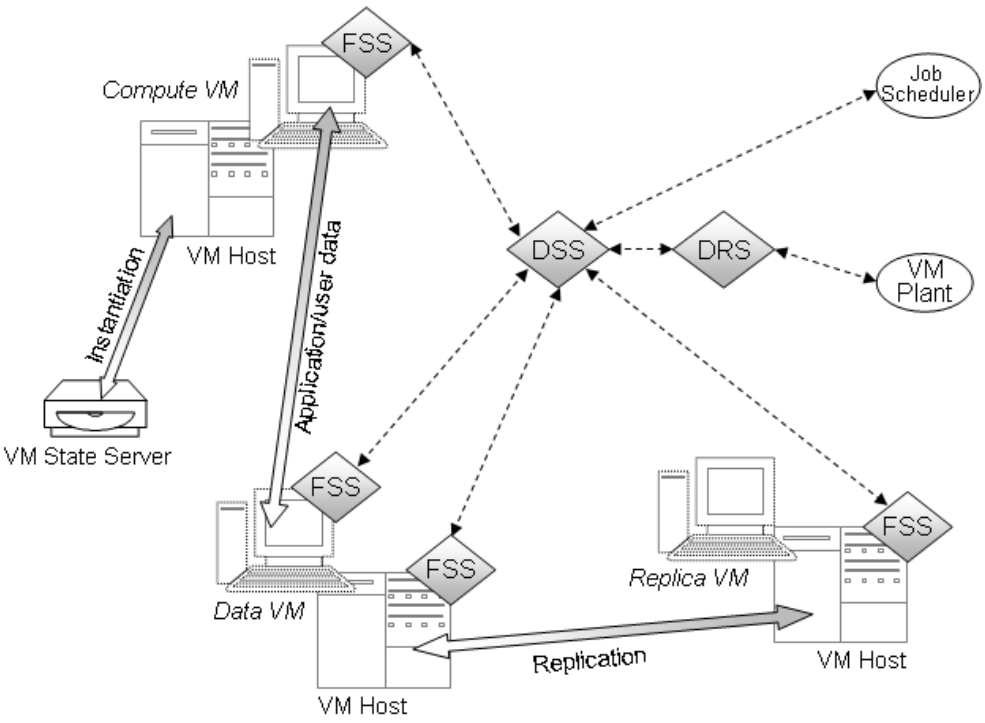

Figure 6-4. The VM-based grid computing system supported by the data management services. To instantiate a compute VM, the VMPlant service requests the DSS to schedule a GVFS session between the VM state server and the VM host. After the VM is started, the job scheduler service can then request the DSS to schedule another session between the compute VM and the data VM, to provide tailored access to application/user data for the job submitted to the compute VM. The DRS also allows for replication of data VMs for improved reliability.

to be provisioned across the network. Previous work has described a VMPlant grid service to support the selection, cloning, and instantiation of VMs [106]. The data management services provide functionality that complements VMPlant to support VM-based grid systems, as depicted in Figure 6-4.

In such a system, the VMPlant service is in charge of managing VMs, including the ones used for computing (execution of applications) and data (storage of application and user data). To instantiate a compute VM for job submission, the VMPlant service requests the DSS to schedule a GVFS session between the VM state server and the VM host, where the VM state transfer can be optimized in the way discussed in Chapter 5. After the VM is started, the job scheduler service can then request the DSS to schedule another

session between the compute VM and the data VM, which provides tailored access to application/user data for the job submitted to the compute VM.

The DRS allows for replication of data VMs for improved reliability. The VM replication can be conveniently done by copying its state files via DSS-scheduled data sessions that use high-throughput transfer mechanisms. The replicated VMs can be distributed across different physical servers and sites to provide tolerance of server and network failures. When a failure happens, the services transparently redirect the application's remote data access to a backup VM and regenerate the lost data VM. To tolerate client-side failures, the services can checkpoint and resume VM instances using the techniques available in existing VM monitors (e.g., VMware suspend/resume, scrapbook UML [117], Xen suspend/resume). With ROW enabled in the GVFS session, buffered data modifications induced by the application execution are also checkpointed as part of the VM's saved state. So, upon failure of the compute VM, the application along with its data changes can be consistently resumed from the last checkpoint.

6.1.5.2 Workflow execution

A workflow typically consists of a series of phases, where in each phase a job is executed using inputs that may be dependent on the outputs of the previous phases. Workflow data requirements can be managed by the DSS with GVFS sessions created on a per-phase basis, and each session can be tailored to suit the corresponding job's needs through the service. In particular, the control over enabling and disabling the consistency protocols and synchronizing client/server data copies is available via the service interface. Hence scheduling middleware can select and steer consistency models throughout the execution of the workflow.

For instance, a typical workflow in Monte-Carlo simulations consists of running large numbers of independent jobs. Their outputs are then post-processed to provide summary results. This two-phase workflow's execution can be supported by the data management services with a data flow (Figure 6-5) such that (1) a session is created

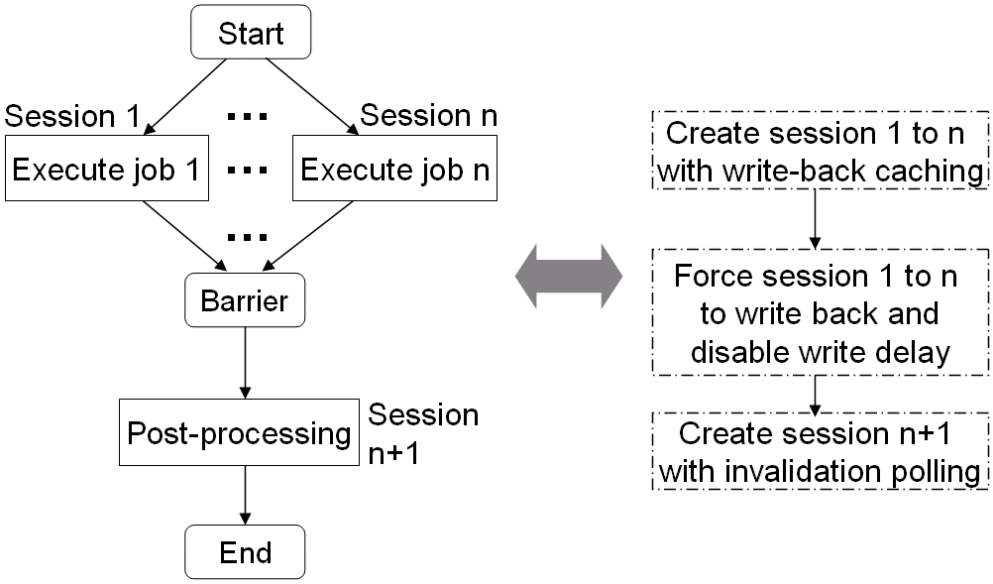

Figure 6-5. A Monte-Carlo workflow and the corresponding data flow supported by the data management services. To support this two-phase workflow's execution, a session is created for each independent simulation job with an individual cache for read/write data; each session is forced to write back and then disable write delay as the simulation jobs complete; and a new session with invalidation-polling consistency is created for running the post-processing jobs that consume the produced data.

for each independent simulation job with an individual cache for read/write data, (2) each session is forced to write back and then disable write delay as the simulation jobs complete, and (3) a new session with the invalidation-polling consistency protocol is created for running the post-processing jobs that consume the data produced in step (1).

Such a workflow can be supported by the In-VIGO system [2][3], where a configuration file is provided by the installer to specify the data requirement and preferred consistency model for each phase. When it is requested by a user via the In-VIGO portal, the virtual application manager interacts with the resource manager to allocate the necessary resources, interacts with the data management services to prepare the required GVFS sessions, and then submits and monitors the execution, for each phase of the workflow.

6.2 Autonomic Data Management

The service-oriented data provisioning and management framework described in the previous sections can serve end-users or job scheduling middleware (e.g., the In-VIGO virtual application manager [2]) to prepare data sessions for application executions. A key challenge faced by such middleware in a grid-style environment is the complexity of managing the performance of many on-demand application instances in the presence of dynamic resource availability. An autonomic application management service [118] is proposed to automatically recover jobs from performance faults based on monitoring and predictions using application execution history and dynamic resource information. The data management services discussed in this dissertation are key to supporting transparent resubmission of jobs, providing fast and on-demand data session establishment to the application management service. On the other hand, the idea of autonomic management can also be applied to the data management in order to reduce its complexity in a grid-style environment.

The use of user-level virtualization to create dynamic, per-application GVFS sessions and the use of service-oriented middleware to control the lifetimes and configurations of the sessions provides the basis for supporting tailored data provisioning to applications in large-scale grid systems. However, the management of GVFS-based data sessions in such an environment is still challenging because of its complexity: large numbers of data sessions need to be created, customized, and terminated on demand based on the applications' lifecycles and requirements; their configurations also need to be timely adapted according to the changes to application workloads and usage of shared processor, network, and storage resources. It is desirable that the data sessions can manage themselves to achieve user or job scheduler expected performance and reliability automatically, so that the management can be simple and the system can be agile.

This section describes the approach to realizing this goal by evolving the data management services into autonomic elements to provide goal-driven self-management of

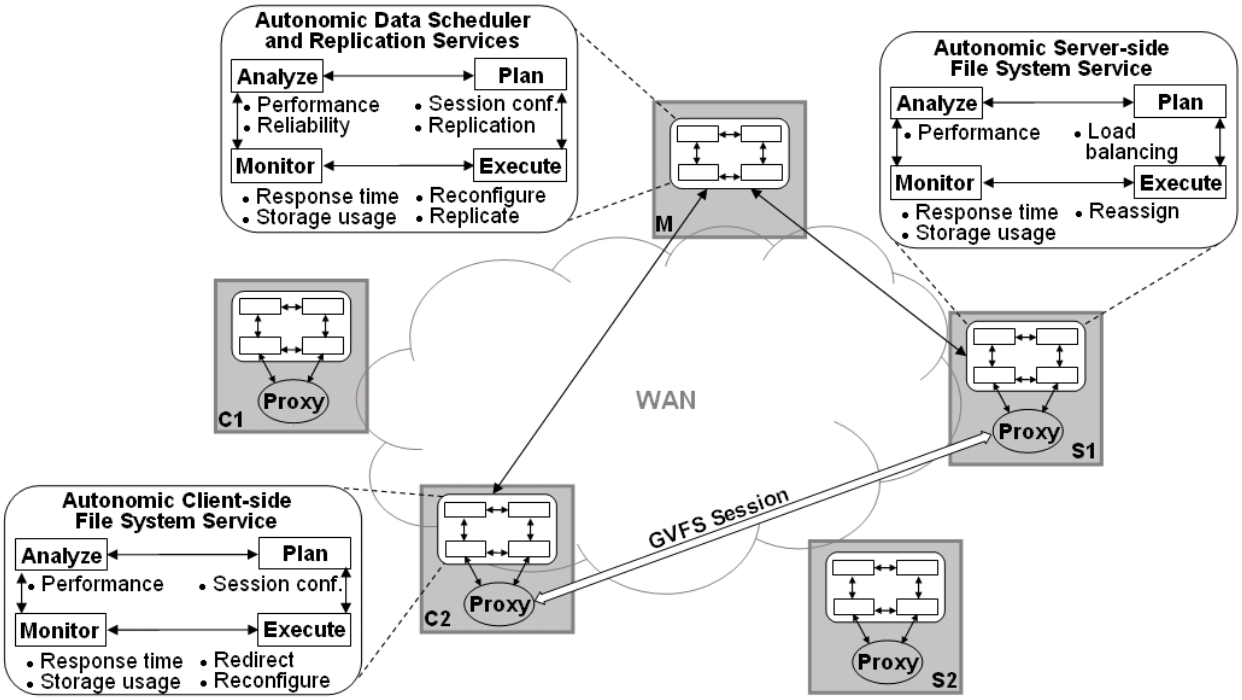

Figure 6-6. Autonomic data management system consists of autonomic data scheduler service, replication service, and client- and server-side file system services. They function as self-managing autonomic elements, which control the client, server, and session of a GVFS data session according to the high-level objectives, and interact with each other to automatically achieve the desired data provisioning behaviors.

the data provisioning (Figure 6-6). The resulting autonomic data management services are capable of automatically monitoring, analyzing, and optimizing the different entities of grid-wide data sessions, as well as cooperatively working together to achieve the desired data provisioning goals. The rest of this section will present the details of these autonomic services as well as an experimental evaluation.

6.2.1 Autonomic Data Scheduler Service

As described in Section 6.1, Data Scheduler Service (DSS) schedules data sessions for application executions. It interacts with Data Replication Service (DRS) to request data replication and interacts with client-side and server-side File System Services (FSSs) to create, configure, and terminate sessions. It is responsible for customizing and isolating different sessions with different configurations. One of the most important session

parameters that can be customized is the size of the client-side disk cache employed by GVFS.

Kernel NFS clients typically buffer data and metadata in memory, but the use of disk caching is rare. In wide-area, long-latency environments, the aggressive use of disk caching can be beneficial to many applications. Therefore, GVFS implements client-side per-session disk caches, which can leverage the large capacity and great persistence of disks to further exploit data locality (Section 4.2.1). However, as the data set size of modern scientific and commercial applications grows rapidly, the DSS needs to carefully manage the storage use for caching, which can have an important impact on the performance of GVFS data sessions.

An application's remote I/O time is estimated by,

$$T = N * r_{mem} * t_{mem} + N * r_{disk} * t_{disk} + N * (1 - r_{mem} - r_{disk}) * t_{network}$$

where N is the total number of remote data requests issued by the application; r_{mem} is the memory buffer hit rate and r_{disk} is the disk cache hit rate; t_{mem}, t_{disk}, and $t_{network}$ are the average service times of a request from memory, local disks, and network storage, respectively.

A data intensive grid application typically has $r_{disk} >> r_{mem}$ and $t_{network} >> t_{disk} >> t_{mem}$, so the hit rate of the disk cache is crucial to delivering good application data access performance. A larger cache results in better hit rate, because capacity misses and conflict misses generally decrease as the cache size grows [92]. However, the relationship between a cache's size and hit rate is a complex one, depending on the locality of data references and the associativity of cache. Therefore, the DSS by default takes a conservative approach in which a session's disk cache is configured with a size larger than its application's data set.

There are also important scenarios where the DSS needs to configure sessions with smaller disk caches. For example, when a node is more powerful or closer to the data server than the other nodes, it is chosen to execute the application because it can provide

better performance even if its storage cannot hold the entire data set. In another common case, multiple applications need to execute on the same node and the available disk space is not enough for their data sets. The DSS can schedule their sessions to run sequentially with full-size disk caches. However, it may be necessary or more beneficial to configure each one with a smaller disk cache and run them concurrently, e.g., in order to meet the deadline requirements or achieve shorter total runtime.

If an application's data access pattern is known from the knowledge base, the DSS can use existing methods [119] to estimate the session's miss rate given the configured cache size, and then estimate its remote I/O performance using the above equation. ($t_{network}$ is monitored online[3] ; N is known from history and is offset by the number of already transferred requests, which is also monitored online.) This information can facilitate DSS to allocate the available storage among multiple sessions that are scheduled to the same node.

A session i's utility U_i represents the value of providing a given level of service to the application. It can be calculated by considering the deliverable session performance and the application priority,

$$U_i = Performance_i * Priority_i$$

where shorter runtime and higher priority generate greater utility value. Since a session's remote I/O time is affected by its disk cache size, given the available storage space as the constraint, the optimal allocation is achieved when the total utility from the different sessions on the node is maximized. The complexity of this optimization computation is bounded by the limited number of concurrent sessions and possible cache sizes. To perform the above analysis, the DSS must monitor the storage usage and the data server

[3] Besides of data requests kernel NFS also issues a considerable number of metadata requests for consistency checks. Most of these requests can be satisfied by GVFS disk caches with its consistency protocols discussed in Section 4.3.

response time on the client. This is realized by interacting with the client-side FSS, which operates an effective monitoring daemon.

Due to the dynamic and non-dedicated nature of grid resources, environment changes may trigger the DSS to reconfigure the session parameters. For example, when the disk usage is reaching the limit because of other local activities, the DSS will detect it and reduce the total space occupied by the caches to avoid overflowing the storage. On the other hand, when more space becomes available for grid use, the DSS can expand caches as necessary.

Note that the changing resource availability may falsify the prediction which has motivated the end-user or application manager to submit a job on this resource. Or, even worse, the client node may crash and fail the job execution. An autonomic application manager should subscribe to these changes and act accordingly by recovering or restarting the job with the assistance from DSS. On the other hand, the server-side fault-tolerance is provided by the autonomic DRS.

6.2.2 Autonomic Data Replication Service

6.2.2.1 Data replication degree and placement

Data replication has long been recognized as the key to achieving high availability. In grid environments replication not only needs to be performed across servers in order to provide failover on server failures, but also should be distributed to different sites to protect against network partitions. However, the limited bandwidth and high delay of WAN make wide-area replication very expensive. Although the DRS uses high-throughput transfer mechanisms (e.g., GridFTP [6]) to replicate data, the overhead is still considerable for large data sets. Because an application's session cannot start until the necessary replicas are ready, this overhead needs to be considered into the cost associated with the session.

The choice of the replication degree, i.e., the number of replicas for a given data set, is a decision that needs to be made based on a benefit-cost analysis. Typically at

least two replicas are required for each data set, so that an application can continue its execution in presence of infrequent failures. As the failure rate goes higher, more replicas are required to sustain good reliability, but the cost from replica creation (and teardown) is also increased. Although it is generally difficult to predict a particular data server's failure rate or MTTF (Mean-Time-To-Failure), it can be estimated based on observation and analysis. Initially every server has a hypothetical failure rate stored in the knowledge base, and it is adjusted and updated by the DRS as failures happen over time. Gradually the value becomes representative of the server's actual reliability.

The available storage capacity for replica placement is shared among the existing sessions. The replicas prepared for a past application execution may also occupy disk space because a lazy style of cleanup is used, where a replica is not removed immediately after its session finishes, in anticipation of future uses of the same data set. Therefore, the storage management takes into account the values of different data sets, in which higher priority applications' data sets have greater values, and live sessions' data sets always value more than those that are not currently in use.

Based on the above considerations, another utility function is also used to solve the replication degree and placement problem. The utility U_d^i of having the ith replica for data set d is computed by the product of the data set's value V_d and the reliability R_d^i provided by the its replicas ($R_d^i = 1 - \prod_{j=1}^{i} r_d^j$, where r_d^j is the failure probability of the data set d's jth replica, decided by the failure rate of the data server where this replica is stored). On the other hand, the cost of creating these replicas is $C_d^i = \sum_{j=1}^{i} c_d^j$, where c_d^j is the overhead from copying the data set to the jth replica's data sever from the nearest existing replica.

Therefore, when considering adding a replica for a data set, its utility as well as the cost and reliability constraints are used to decide whether to add it and where to place it,

as follows:

$$U_d = V_d * R_d^i$$

$$R_d^i \geq R_{\min}$$

$$C_d^i \leq C_{\max}$$

where R_{min} is the desired minimum level of reliability for the data set, and C_{max} is the maximum tolerable replica creation overhead. This algorithm tends to place a replica on more reliable servers when the reliability is more important, and put it on closer servers if the cost is more concerned. When multiple allocations are available, the DRS chooses the server that has the best fit available space. If replacement is necessary due to the lack of storage, the replica with the lowest utility is chosen and evicted.

6.2.2.2 Data replica regeneration

Autonomic replica regeneration is also supported by the DRS. When a data server failure occurs, the running sessions' replicas on that server need to be promptly regenerated in order to minimize the time windows in which the necessary replication degrees are not satisfied for the data sets. In this process, human-intervention should be avoided because it tends to be slow and costly. Furthermore, the impact of failures on applications should also be reduced to the minimum; it is desirable that applications can continue their executions without interruptions. This is realized by the autonomic FSS and will be discussed shortly.

The DRS achieves autonomic replica regeneration by means of automatic failure detection and replica reconfiguration. A failure is usually notified by the DSS which monitors the session through the client-side FSS. However, a failure reported by the client-side FSS can be caused by network partitioning between the client and server. This is confirmed by the DRS if it can still connect to that server, and then only the data sets that are used by this particular client need to be regenerated. Once a failure is determined, the DRS immediately allocates storage space using the aforementioned algorithm and regenerates the lost replicas for the running sessions on the newly selected

data servers. The information about these new replicas is also informed to the DSS, so that it can reconfigure the concerned sessions to use them in need of failover.

6.2.3 Autonomic File System Service

6.2.3.1 Client-side file system service

Client-side FSS facilitates the task of autonomic DSS by monitoring storage usage and server response time, and executing the session configurations decided by the DSS. As discussed in Section 6.2.1, a FSS controls the proxy to shrink or expand a session's disk cache as instructed by DSS. A disk cache is structured as file banks that contain data blocks hashed according to their file handles and offsets. Sophisticated algorithms can be conceived to reduce a cache's size by evicting the least recently used blocks and rehashing the other ones, which often incurs substantial overhead. Instead, the proxy removes the least-used and most-clean file banks from the cache until the required shrinkage is achieved. This needs only a simple re-mapping of file banks but not any rehashing of data blocks, and the new cache size can immediately take effect.

Client-side FSS is also the key to realizing application-transparent failover in presence of server-side failures, including data server failure, network partitioning between the client and server, and server or network overloading. As discussed in Section 4.5.3, these are detected when a major timeout (e.g., 100 times of the average response time) occurs to a data request. In order to recover from a fault, the proxy immediately redirects the session to the backup data server. The FSS also reports a detected fault to the DSS so that it can ask the DRS to take actions on replica regeneration.

To achieve failover without any interruption to the application, a session uses an active-style model, as presented in Section 4.5.2, to maintain data consistency among the replicas. Every data modification request issued by the application is multicast from the proxy client to the session's every proxy server in a reliable manner. Consequently, each replica has the exact same copy of the data set during the entire session, and if any

of them crashes, it has no impact upon the application since the remaining replicas can continue to service as usual.

Although it is necessary to write-all, a session can choose to use read-all or read-one. In the former case read operations are also performed on all the replicas. This model is employed to further improve reliability against not only server crashes, but also Byzantine failures, in which the proxy client collects and compares all the replies from the proxy servers and then decides on the correct one. However, the disadvantage of this model is that it limits the performance of the session to the slowest replica server all the time. On the other hand, it is often safe to assume that a successful data operation is correct because there are other mechanisms from hardware to software that are in place and can promise that an error would not happen without being noticed. Read operations are the most common ones for typical applications, and thus the most valuable to optimize to achieve speedup according to Amdahl's Law [92]. Therefore, GVFS sessions typically employ this "read-one, write-all" replication model.

This model relies on client-side FSS' autonomic functions to choose the best replica server to perform read operations throughout a session. The chosen server is called the *primary server* for the session. It is decided from the session's replica servers based on the performance that can be delivered for the application's remote I/O operations. The primary server can change over time as network conditions and server loads vary. So the FSS monitors the performance of the replica servers periodically using a monitoring daemon.

It is important to design an accurate and low overhead mechanism to measure the performance. From a session client's point of view, the response time of a remote data request is determined by the network delay as well as the data server's CPU processing delay and disk access delay. Simple network probing mechanisms, e.g., ping, can give information about the network's performance, but not the server's. Using NULL RPC requests to the server incurs little overhead and can measure both the first and second

delays, but it cannot reveal the server's I/O load and disk performance. Instead, the monitoring daemon issues very small writes on the GVFS partition and uses the response times to estimate the performance.

Writes are used in the measurement because it is difficult to prevent the effect of server-side caching (processor caching and memory buffering) with read operations. The FSS monitoring daemon periodically performs a one-byte write at a different block of a hidden remote file, and it requests the server to commit the write so as to avoid the effect of server-side write delay. Even though the hidden file's size may reach a large value for a long session, it does not necessarily occupy much space on disk because the server's file system typically uses holes on sparse files. To further reduce this overhead, the monitoring daemon automatically truncates this file and starts over from beginning after a few hundreds of probes.

Because this monitoring mechanism has very low overhead and it is done outside of the proxy that is responsible for processing application's data requests, the session's performance is almost intact. The FSS can use well-known time series analysis algorithms to predict a replica server's future performance based on the observed response times, and then to make decisions on the primary server selection. Complex algorithms are not suitable because they would intensively compete for CPU with the running applications and not necessarily give the best predictions. On the other hand, simpler algorithms have been proven effective in many cases [120]. In this prototype the exponential smoothing algorithm [121] is used. Once the primary server is planed to change, the FSS controls the proxy to switch it transparently to the application.

6.2.3.2 Server-side file system service

An autonomic server-side FSS monitors the server's storage usage, which helps the autonomic DRS to decide replica placement. It also monitors the response times of RPC requests forwarded by a proxy server to the kernel server. The server is not necessarily on the proxy's localhost, but can be a virtualized server that consolidates network

attached devices [122]. There are well-studied algorithms [123] to provide load balancing in this scenario which can be leveraged by the FSS and hence it is not discussed in this dissertation.

6.2.4 Evaluation

6.2.4.1 Setup

A prototype for the proposed autonomic data management system has been implemented and its autonomic features are evaluated in this section. VMware-based virtual machines were used to setup the file system clients and servers, which were hosted on two physical cluster nodes. Each physical node has dual 2.4GHz hyper-threaded Xeon processors and 1.5GB memory. Each VM was configured with 64MB memory and was installed with SUSE LINUX 9.2. The emphasis of the experiments is in wide-area environments, which were emulated using NIST Net [95]. Unless otherwise noted, every link was configured with a typical wide-area RTT of 40ms.

The experiments were conducted by using a typical file system benchmark, IOzone [88], to represent the I/O part of grid applications. It was executed on the clients with input accessed from the servers via GVFS. The GVFS sessions were virtualized upon NFSv3, using 32KB data block transfer, and the data servers exported the file system without write delay and with synchronous access. Every experiment was started with cold kernel buffer and GVFS disk caches by unmounting the file system and flushing the disk cache.

6.2.4.2 Autonomic session redirection

In a grid environment network latency and throughput are often affected by the existence of parallel TCP transfers [124], for example, the popular use of GridFTP [6]. Figure 6-7 shows the latency between two nodes under such influence in a real wide-area setup. Each node was located in a different 100Mbps LAN and there were parallel TCP transfers (from the use of Iperf [89]) between another pair of nodes from these two LANs.

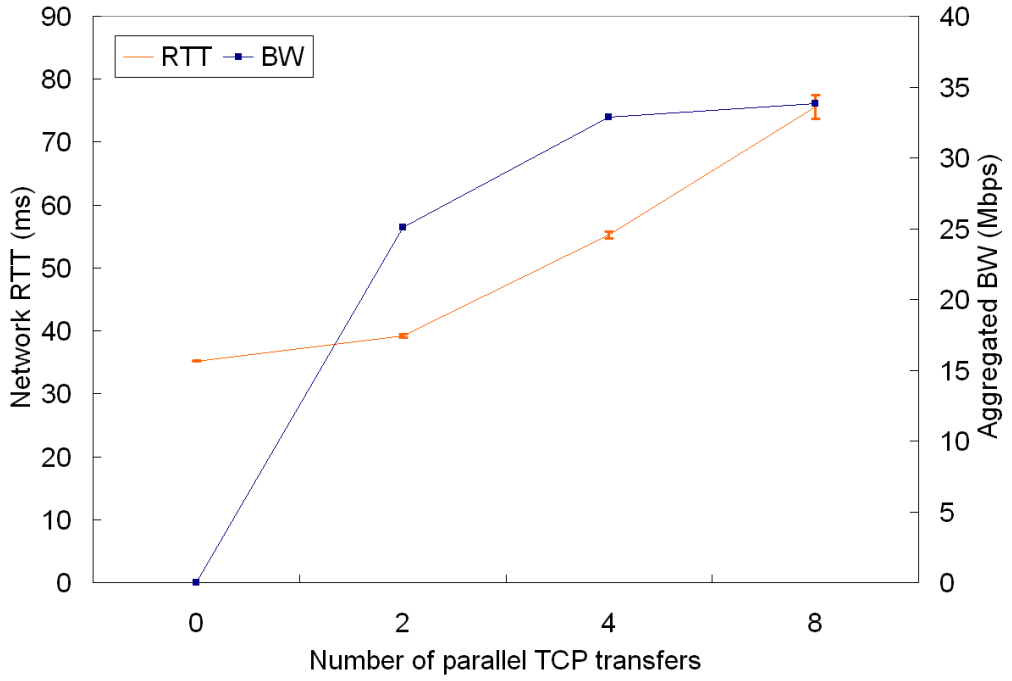

Figure 6-7. Network round-trip time (RTT) between two WAN nodes impacted by third-party parallel TCP transfers. The aggregate bandwidth (BW) consumed by the interfering parallel TCP transfers is also shown in the figure.

The data demonstrate that the latency grows rapidly as the number of parallel TCP streams used by the third-party transfer increases.

Such a scenario was emulated in the experiment and used to study the effectiveness of autonomic session redirection in the presence of network performance fluctuations. The data set was replicated across two servers, and the client was connected to servers (server 1 and 2) via two independently emulated WAN links, where each link's latency was varied randomly with the values shown in Figure 6-7 in existence of 0, 2, 4 and 8 parallel TCP streams, and with decreasing probabilities for these values.

The IOzone benchmark was executed on the client node, which reads and rereads a 256MB file accessed through GVFS with different data server configurations: using *server 1* statically; using *server 2* statically; and using *autonomic session redirection* between these servers dynamically. The average server response times were collected every 10s throughout the execution of the benchmark, as shown in Figure 6-8(a). The results show

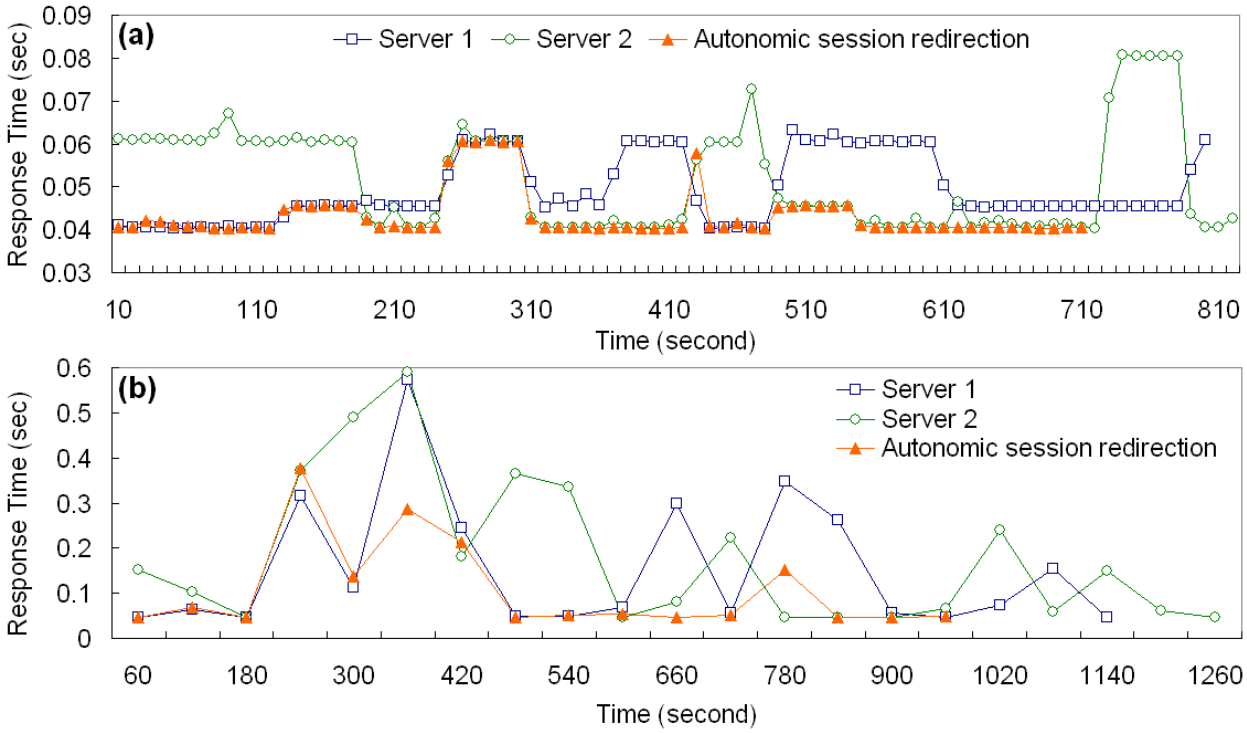

Figure 6-8. Average response times during the execution of IOzone (read/reread mode) with a 256MB input accessed through GVFS with different data server configurations: using *server 1* statically; using *server 2* statically; and using *autonomic session redirection* between these servers dynamically. The benchmark was executed under dynamic network load fluctuations in (a) and dynamic server load variations in (b).

that with autonomic redirection the GVFS session can almost always choose the better link for data access, and consequently the runtime of the benchmark is about 13% better than using server 1 statically and 16% better than using server 2 statically.

The second scenario considers the effect of dynamic server load variations on the performance of a session. The I/O-intensive jobs (executions of IOzone with read/reread of different 256MB input files) were loaded to the server following a Poisson process, and the intensity was varied by randomly choosing the number of concurrent jobs between 0 and 6. Then the benchmark was executed with the same configurations as above. The average server response times were collected every 60s throughout the execution and are plotted in Figure 6-8(b). The results also show that autonomic redirection can achieve the best

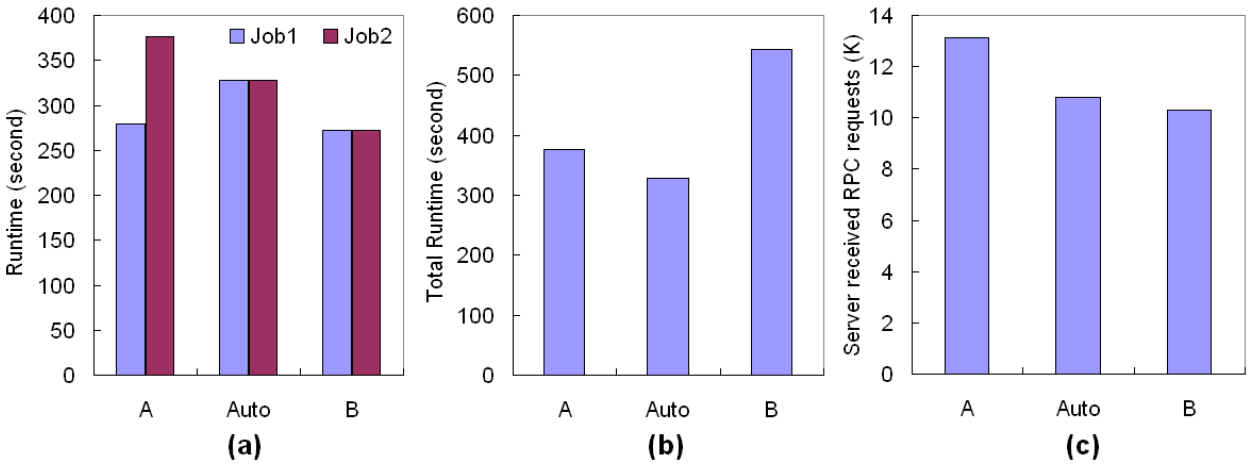

Figure 6-9. (a) Runtimes of two executions of IOzone (job 1 and 2) which randomly read a different 256MB file through GVFS; (b) their total runtime, and (c) total number of requests received by the server. Different cache configurations are used for the sessions: *A*, starts the first one with full-size cache and the second one without caching, concurrently; *B*, starts them sequentially with full-size caches; *Auto*, starts them concurrently and divides the available storage autonomically between their caches.

response time, and helps the benchmark to run 16% faster than using server 1 statically, and 29% faster than using server 2 statically.

6.2.4.3 Autonomic cache configuration

The second experiment studies the use of autonomic cache configuration by the DSS while scheduling different data sessions. In this setup, two tasks need to be run on the same client node, where each task executes IOzone with random reading of a different 256MB file accessed from the data server via GVFS. The DSS is required to prepare two data sessions for these tasks but the available storage on the client can only hold 256MB of disk caches. So three different configurations are possible for these sessions: *A*, starts the first session with full-size cache and the second one without caching, concurrently; *B*, starts them sequentially with full-size caches; and *Auto*, starts the sessions concurrently and splits up the available storage autonomically between the sessions' caches based on their utilities (in this setup each session gets half of the disk space because they are assumed to be equally important).

Figure 6-9 shows the runtimes of the jobs as well as their total runtime and total number of requests received by the server during their executions. (For concurrent sessions, the total runtime is the maximum one among the individual runtimes.) Compared to A, the autonomic configuration provides better fairness between the jobs and it also greatly reduces the server load (the amount of server-received requests is down by 18%); compared to B, the autonomic configuration's total runtime is much shorter (reduced by 40%).

6.2.4.4 Autonomic data replication

The last experiment investigates the autonomic data replication in presence of server-side node or network failures. A sequence of tasks were launched on the client node sequentially, where each one ran IOzone in random reading mode with 512MB input accessed from the data servers through GVFS. The data servers failed randomly, where the failures were modeled as a Poisson process with an average inter-arrival time of half an hour. The experiment used a replication degree of 2 for the data sets, and failures were injected on the servers by randomly choosing one of them to stop its network connection. Two different situations were considered for the tasks: *independent*, each task had an independent data set (the input file) and was scheduled with a different data session; *dependent*, the tasks had the same data set and shared the same data session.

Figure 6-10 shows the timeline of events as they happened during the experiment. In the independent tasks case, a total of four server-side failures happened. Each failure caused a new replica to be generated by copying the data from the remaining server to a new server. Two of the failures occurred at the primary data server and also triggered the client to redirect the session's data access to the backup server. These all caused delays in the benchmark's executions, e.g., the second run took the longest time to finish because two failures occurred during that run. Nonetheless, every run successfully completed regardless of the failures. In the dependent tasks case, the disk cache was warmed up after the first run, which not only substantially reduced the times of the following runs,

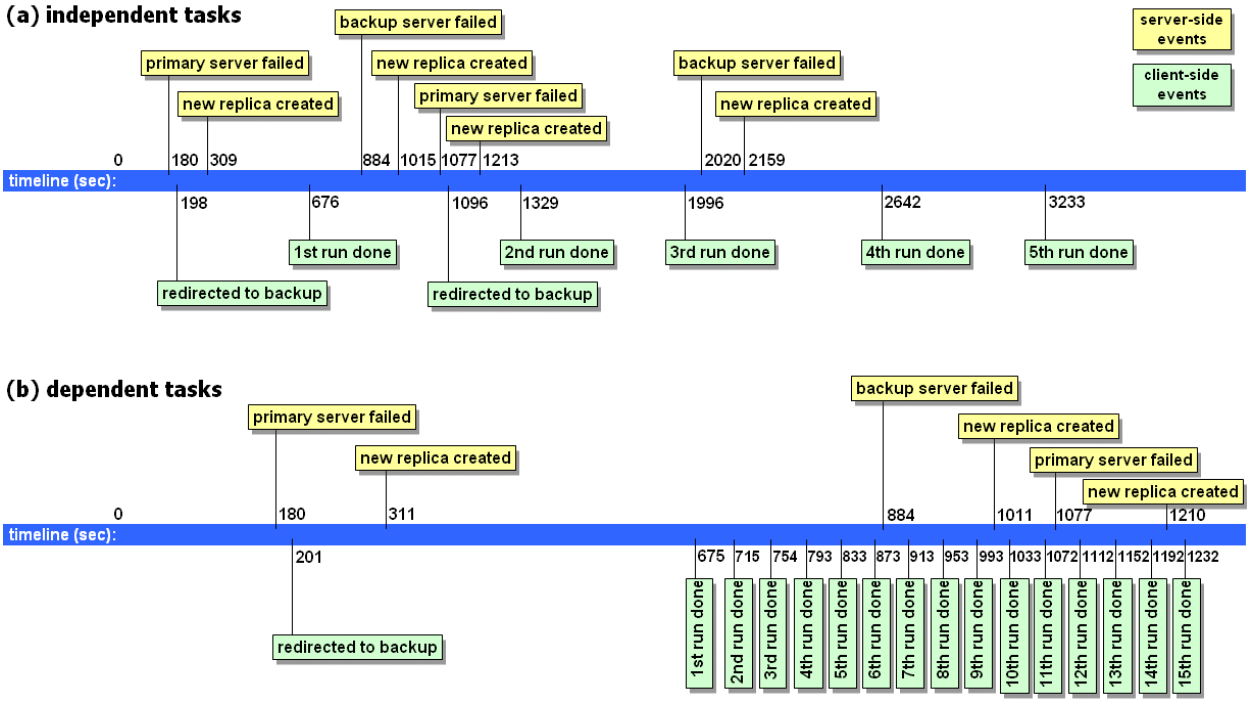

Figure 6-10. Timeline of events happened during a sequence of executions of IOzone. Each run randomly read a 512MB input accessed from the data servers through GVFS. The data servers failed randomly, and replica regenerations were triggered accordingly. In (a), the executions were independent and were supported by different GVFS sessions; in (b), the executions used the same data set and shared the same session.

but also made the client completely unaware of the server-side failures and delays. This experiment demonstrates that by using the autonomic services to select and combine application-tailored enhancements, such as caching and replication, the performance and reliability of grid-wide data sessions can be substantially improved in an automatic and transparent manner.

CHAPTER 7
CONCLUSION

7.1 Summary

This dissertation focuses on data management in a grid-style environment, where applications and data are distributed on resources across administrative boundaries and wide-area networks. Such an environment is becoming increasingly typical as the scale of today's large distributed computing systems, such as scientific grids, enterprise data centers, and "cloud" computing systems, grows. The dissertation has designed and implemented a novel system for data management in these environments, and it advocates the three key elements of this solution: user-level distributed file system (DFS) virtualization, application-tailored data provisioning, and autonomic service-based management.

Several types of grid data management approaches are available to provide cross-domain data access to applications. Fundamentally, they differ in the level where remote data access is introduced in the system and in the degree of transparency provided to the system. Application-level approaches explicitly involve data management middleware in staging files for applications, and thus they are strongly tied to specific applications. Such approaches are not application transparent, which makes it difficult to enable a wide variety of applications to harness the power of a large distributed computing system. Operating system (O/S) level approaches achieve application transparency by implicitly service an application's I/O requests with remote data access. But these approaches are not O/S transparent: they require O/S-specific modifications which are difficult to deploy on the typically heterogeneous and non-dedicated resources in a grid-style environment.

User-level approaches can provide transparency to both applications and O/Ss, and one of such approaches is based on intercepting an application's library calls or system calls and mapping them to remote data access. However, this approach is only applicable to applications that can be relinked or O/Ss that have system call

tracing capability. Another user-level approach, which is taken by this dissertation, is through the interception and handling of DFS calls, in essence, virtualizing an O/S-level DFS and providing grid-wide virtual file systems (GVFSs). It preserves the generic file system interface to provide complete application transparency and leverages widely-available DFSs (e.g., NFS) to achieve great O/S transparency. Moreover, based on this virtualization, enhancements to grid-wide data access can be also realized without changing applications and O/Ss, and the management of data provisioning can be decoupled from the underlying physical resources and optimized independently.

A unique contribution made by this dissertation is on application-tailored data provisioning. None of the existing solutions, including the closely related DFS-based ones, can address the diverse needs of applications. Nonetheless, the inefficiency, insecurity, and unreliability of grid-style environments require application-tailored optimizations on different aspects of remote data access. This dissertation addresses this need by proposing user-level enhancements for GVFS on performance, consistency, security, and fault-tolerance. GVFS-based data sessions are created on demand on a per-application basis, and they can independently select, customize these enhancements according to its application's characteristics and requirements.

This dissertation is also the first to realize the importance of applying autonomic techniques to data management in grid-style environments, and addressing the complexity of managing large numbers of on-demand data sessions with changing application workloads and resource availability. It has developed a novel autonomic data management system based on GVFS virtualization and service-oriented management. A set of services are developed to provide flexible control of the lifecycles and configurations of GVFS data sessions, and to support interoperable interactions with other middleware services based on the Web Service Resource Framework (WSRF). Intelligence can then be built into these services to enable automatic configuration, optimization, healing, and protection of the data provisioning in accordance with high-level objectives.

Every aspect of the proposed system has been thoroughly evaluated with experiments based on typical file system benchmarks and real applications. A key insight from the results is that the overhead from virtualization is not significant, but it can enable important improvement and functionality which are not available or possible in the underlying physical system. On the other hand, the virtualized systems can leverage these enhancements to further reduce the overhead and even outperform the physical system. Specifically, GVFS virtualization enables dynamic data provisioning and flexible application-tailored enhancements to address the limitations of traditional DFSs (e.g., NFS) in grid-style environments. In addition to providing strong consistency, security, and reliability to grid-wide data access, the enhanced GVFS is able to not only hide the overhead from user-level proxy based virtualization, but also deliver significant speedup compared to NFS in WAN.

These experiments also demonstrate the effectiveness of using the proposed data management services to create and terminate GVFS sessions on demand as well as to customize and apply the various enhancements on the sessions as needed. These services also allow the interactions with other WSRF-based middleware, e.g., services built with Globus Toolkit [69], and support its data management requirements. The autonomic data management architecture described in this dissertation enables autonomic functions to be developed for self-management on different aspects of data provisioning. Experimental evaluation on the provided several autonomic features show that they can automatically allocate resources to GVFS sessions based on policies, improve data access performance under network and server load fluctuations, as well as protect applications and regenerate data replicas in the presence of dynamic server failures.

This GVFS-based approach has been applied to support the use of virtual machines (VMs) as execution environments in grid computing, in which on-demand access to both user data and VM state is transparently provided to the applications and VMs dynamically instantiated across grids. Results from experiments with VMware-based

VMs show that this GVFS solution supports fast VM instantiations via cloning and efficient executions of applications in VMs. It substantially reduces the overhead of provisioning VMs across WAN, compared to the approaches based on native NFS or full-file download/upload.

An implementation of this dissertation's approach has also been deployed in the production In-VIGO system [2][3], a grid system for scientific applications and the prototype of nanoHUB [125]. It has supported applications from many different disciplines, such as the spectroscopy study for biomedical scientists [91] and storm surge modeling for costal researchers [102]. The GVFS solution provides transparent, on-demand data access to not only these applications enabled by In-VIGO, but also the other middleware components of the In-VIGO system, such as the user interface manager, virtual application manager, and VM manager. In the past a few years, tens or even hundreds of GVFS sessions have been dynamically created everyday to support the application executions on grid resources and the smooth functioning of In-VIGO.

7.2 Future Work

The fundamental goal of my research is to design and develop DFS virtualization and services for flexible and scalable data management in grid-style environments. This dissertation has made substantial progress towards this goal, and it has also laid a solid foundation for future research. Further improvement of the proposed data management system can be considered along the following three directions.

7.2.1 Performance

Distributed file system based data provisioning is important to supporting application transparency, and block-based data transfer is very efficient for interactive applications as well as large sparse file access. However, for large bulk data transfer, the proposed data management system has to rely on additional high-throughput data transfer mechanisms such as GridFtp [6]. This inefficiency of block-based DFS access can be attributed to two factors: conservative prefetching policies and low-throughput network data transport

mechanisms. Future research can be conducted to improve GVFS for bulk data transfer by specifically addressing these two problems.

Traditional DFS clients use only conservative prefetching policies, e.g., the adaptive read-ahead algorithm in Linux kernel uses a relatively small value for the read-ahead window size. These policies are designed under the assumptions for LAN environments, where the penalty of misprediction on prefetching often outweighs the benefit from aggressive prefetching, because the network round-trip time (RTT) is relatively small and clients use limited memory for caching. In a grid-style environment, these assumptions do not hold any more. Wide-area networks have high latency as well as large bandwidth-delay product (BDP — the product of the link capacity and the RTT of a packet). With the emergence and spread of new WAN technologies, network bandwidths are getting higher, but the latencies are still bounded by the speed of light, which result to even larger network BDP. If the network bandwidth is underutilized, there is little difference in the cost of fetching a larger block of data than a smaller one, but there would be a significant gain in performance if the extra data are indeed useful to the client. Therefore, it would be beneficial to use aggressive prefetching to fill the "pipe" up and take advantage of the available bandwidth.

If a client-side cache's capacity is limited, premature prefetching could push other useful data out of the cache, increase cache misses, and cause performance degradation. This is also part of the reason why traditional memory-caching only DFSs restrict the amount of prefetching. The GVFS approach employs disk caches, which have much larger capacity (in the order of gigabytes or even terabytes) than physical memories (in the order of megabytes), and thus have much less concern on the caches being flooded. Moreover, disk caching is persistent in face of client crashes/reboots, which means that prefetched data can be useful for the client over a long period of time. Based on these reasons, aggressive prefetching with GVFS disk caching has the potential to improve the performance of remote data access substantially.

The GVFS approach leverages NFS remote procedure calls (RPC) for remote data access, which can be transferred over either UDP or TCP. Traditional NFS implementations use UDP because they are designed for use in the relatively reliable LAN environments. To cope with packet loss, NFS clients have to retransmit the requests that have timed out, which, however, can significantly impact the throughput on WAN. If the timeout is too short, the retransmissions may aggravate the problem such as network congestion or server overloading. If the timeout value is too large, the client may be unnecessarily idle waiting for a lost packet and cause additional latency for the data request. The NFS implementations based on TCP perform better over a long-latency or unreliable network because TCP provides more efficient reliable data transfer at the transport layer. Nonetheless, it is still difficult to achieve high-throughput in WAN with TCP, due to its inefficiencies in networks that have high BDP [126], including the slow congestion-control algorithm, the bandwidth underutilization in face of errors that are not caused by congestion, and the bias against TCP flows with higher RTTs.

To address the above limitations of native UDP and TCP protocols, future research can study high-throughput data transport to serve both remote data access and data prefetching. In addition to improving performance, it is important that such transport can be transparently deployed on resources in grid-style environments. It should not require any modifications to existing network infrastructures (e.g., O/S networking stack), and it should be convenient to install without local administration involvement on the systems. Therefore, an application-level and user-level transport enhancement on GVFS is more desirable.

7.2.2 Intelligence

In order to deal with the data management complexity and provide optimal data service, an autonomic grid data management system is proposed in this dissertation based on GVFS and its management services. It supports policy-driven self-management of GVFS sessions in several aspects, including cache configuration, data replication, and

session redirection. Working towards a more advanced autonomic grid data management system, more self-optimization features can be developed and integrated into this system.

In particular, cache replacement and prefetching decisions can have significant impact on data access performance. Current GVFS implementation employs a simple priority-based cache replacement policy, in which dirty cache blocks have higher priority than clean blocks, and within the same priority class a block is randomly chosen to evict. Results from the experimental evaluation show that this simple scheme can deliver good performance by taking advantage of data locality. As a first-step enhancement to this simple scheme, traditionally successful cache replacement algorithms (e.g., LRU, MRU) can be employed. However, it can be further improved by adopting more intelligent policies based on the prediction of application data access patterns.

While studying these patterns, it is important to maintain application-transparency, which has always been the top design goal of the proposed data management system. Therefore, approaches that rely on explicit hints produced by applications (e.g., [127]) are not appealing to this research. Section 5.2 has described the use of metadata handling to capture application-specific knowledge and optimize data transfer, which is employed to support efficient VM state transfer. Such an application-transparent approach can be extended to a more comprehensive framework, in which the knowledge about application data access characteristics and requirements is automatically learned and leveraged by the autonomic data management system. Future research can investigate how to achieve this goal based on techniques such as trace tracking and analysis, offline and online learning, as well as phase and patten classification.

7.2.3 Integration

By the virtue of transparency, the proposed data management approach is able to not only support the heterogeneous systems in grid-style environments, but also exploit the strengths of the diverse storage assets deployed in the systems, such as storage area network (SAN), parallel file system, and RAID systems. Therefore, in future research, a

scalable data management middleware system can be built based on extending GVFS and its management services, and bridging existing, special-purposed file and storage systems.

The GVFS will be the backbone in the data management, connecting heterogeneous resources in different domains, and providing application-tailored, transparent data access across WAN. Specifically, it can be extended to support parallel file access for high-throughput, by bridging a parallel file system, and support storage networking for high-performance, by bridging a SAN file system. The management services will be the brain to control GVFS as well as the bridged systems and to optimize the end-to-end data provisioning. They can be extended to support more high-level functionalities (e.g., global snapshots, live migration), and to provide scalable management in a cooperative, peer-to-peer manner.

Based on this approach, the grid VM data management proposed in this dissertation can be also extended to support an even larger-scale high-performance and high-throughput computing system built upon virtualized resources across data centers, enterprises, and grids. In this envisioned system, the VM data management can enable fast cloning and versioning on VM state servers to provide time- and space-efficient VM creation and customization, and employ smart prefetching, caching, and checkpointing on the VM hosts to achieve efficient and reliable VM instantiations and executions.

REFERENCES

[1] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *Int. J. High Perform. Comput. Appl.*, vol. 15, pp. 200–222, August 2001.

[2] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu, "From virtualized resources to virtual computing grids: the in-vigo system," *Future Generation Computer Systems*, vol. 21, pp. 896–909, June 2005.

[3] A. M. Matsunaga, M. O. Tsugawa, M. Zhao, L. Zhu, V. Sanjeepan, S. Adabala, R. J. O. Figueiredo, H. Lam, and J. A. B. Fortes, "On the use of virtualization and service technologies to enable grid-computing," in *Proc. Euro-Par*, pp. 1–12, 2005.

[4] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten, "Portable batch system: External reference specification," tech. rep., MRJ Technology Solutions, November 1999.

[5] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," in *Proc. the Workshop on Job Scheduling Strategies for Parallel Processing*, (London, UK), pp. 62–82, Springer-Verlag, 1998.

[6] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster, "Secure, efficient data transport and replica management for high-performance data-intensive computing," in *Proc. the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, (Washington, DC, USA), IEEE Computer Society, 2001.

[7] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke, "Gass: A data movement and access service for wide area computing systems," in *Proc. the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, (Atlanta, GA), pp. 78–88, ACM Press, 1999.

[8] "Gt 4.0 reliable file transfer (rft) service." URL: http://www.globus.org/toolkit/docs/4.0/data/rft/.

[9] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *Proc. 8th International Conference on Distributed Computing Systems*, (San Jose, CA, USA), pp. 104–111, June 1988.

[10] J. Bent, D. Thain, A. A. C. Dusseau, A. R. H. Dusseau, and M. Livny, "Explicit control in a batch-aware distributed file system," in *Proc. the 1st USENIX Symposium on Networked Systems Design and Implementation*, 2004.

[11] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman, "Ufo: A personal global file system based on user-level extensions to the operating system," *ACM Transactions on Computer Systems*, vol. 16, no. 3, pp. 207–233, 1998.

[12] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data-intensive computing," in *Proc. the Workshop on Adaptive Grid Middleware*, September 2003.

[13] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "Legionfs: a secure and scalable file system supporting cross-domain high-performance applications," in *Proc. the 2001 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 59–59, ACM Press, 2001.

[14] J. Bent, V. Venkataramani, N. Leroy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Flexibility, manageability, and performance in a grid storage appliance," in *Proc. the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, (Washington, DC, USA), IEEE Computer Society, 2002.

[15] N. H. Kapadia and J. A. B. Fortes, "Punch: An architecture for web-enabled wide-area network-computing," *Cluster Computing*, vol. 2, no. 2, pp. 153–164, 1999.

[16] D. Thain, J. Basney, S. C. Son, and M. Livny, "The kangaroo approach to data movement on the grid," in *Proc. the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC-10)*, pp. 7–9, 2001.

[17] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes, "A case for grid computing on virtual machines," in *Proc. the 23rd International Conference on Distributed Computing Systems*, (Washington, DC, USA), IEEE Computer Society, 2003.

[18] M. Kozuch and M. Satyanarayanan, "Internet suspend/resume," in *Proc. the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, (Washington, DC, USA), IEEE Computer Society, 2002.

[19] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," in *Proc. the 17th USENIX conference on System administration*, (Berkeley, CA, USA), pp. 181–194, 2003.

[20] M. Zhao, J. Zhang, and R. Figueiredo, "Distributed file system support for virtual machines in grid computing," in *Proc. the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, (Washington, DC, USA), pp. 202–211, IEEE Computer Society, 2004.

[21] B. Callaghan, *NFS Illustrated*. Addison-Wesley, 2002.

[22] "Nfs: Network file system protocol specification." RFC 1094, March 1989.

[23] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," in *Proc. USENIX Summer*, pp. 137–152, 1994.

[24] P. J. Leach and D. C. Naik, "A common internet file system (cifs/1.0) protocol." Internet Draft, 1997.

[25] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, pp. 51–81, February 1988.

[26] "Open source version of afs." URL: http://www.openafs.org.

[27] P. J. Braam, "The coda distributed file system," *Linux J.*, vol. 1998, no. 50es, 1998.

[28] B. Callaghan and T. Lyon, "The automouner," in *Proc. the Winter 1989 USENIX Conference*, pp. 43–51, 1989.

[29] M. Blaze, "A cryptographic file system for unix," in *Proc. 1st ACM Conference on CCS*, pp. 9–16, 1993.

[30] K. Fu, F. M. Kaashoek, and D. Mazieres, "Fast and secure distributed read-only file system," *ACM Trans. Comput. Syst.*, vol. 20, pp. 1–24, February 2002.

[31] "Cache file system (cachefs)." White-Paper, Sun Microsystems, Incorporated, February 1994.

[32] V. Srinivasan and J. Mogul, "Spritely nfs: experiments with cache-consistency protocols," in *Proc. the twelfth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 44–57, ACM Press, 1989.

[33] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the sprite network file system," *ACM Trans. Comput. Syst.*, vol. 6, pp. 134–154, February 1988.

[34] R. Macklem, "Not quite nfs, soft cache consistency for nfs," in *Proc. the USENIX Winter 1994 Technical Conference*, (San Fransisco, CA, USA), pp. 261–278, 1994.

[35] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Rfc3530: Network file system (nfs) version 4 protocol." URL: http://www.ietf.org/rfc/rfc3530.txt, 2003.

[36] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the pangaea wide-area file system," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 15–30, 2002.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "Oceanstore:

an architecture for global-scale persistent storage," in *Proc. the ninth international conference on Architectural support for programming languages and operating systems*, vol. 28, pp. 190–201, ACM Press, December 2000.

[38] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The oceanstore prototype," in *Proc. the Conference on File and Storage Technologies*, USENIX, 2003.

[39] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the world wide web," in *USENIX Symposium on Internet Technologies and Systems*, 1997.

[40] D. E. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. USA: Morgan Kaufmann Publishers, Inc., 1999.

[41] J. Linn, "The kerberos version 5 gss-api mechanism." RFC 1964, June 1996.

[42] "Pki." URL: http://www.oasis-pki.org/resources/techstandards/.

[43] "Public-key infrastructure (x.509) (pkix) charter." URL: http://www.ietf.org/html.charters/pkix-charter.html.

[44] M. Eisler, A. Chiu, and L. Ling, "Rpcsec_gss protocol specification." RFC 2203, September 1997.

[45] J. Linn, "Generic security service application program interface." RFC 1508, September 1993.

[46] M. Eisler, "Lipkey - a low infrastructure public key mechanism using spkm." RFC 2847, June 2000.

[47] P. Honeyman, W. A. Adamson, and S. Mckee, "Gridnfs: Global storage for global collaborations," *Local to Global Data Interoperability - Challenges and Technologies*, 2005.

[48] "Secure nfs via ssh tunnel." URL: http://www.math.ualberta.ca/imaging/snfs/.

[49] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," *ACM SIGOPS Operating Systems Review*, vol. 34, no. 2, pp. 19–20, 2000.

[50] M. Kaminsky, G. Savvides, D. Mazieres, and F. M. Kaashoek, "Decentralized user authentication in a global file system," *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 60–73, December 2003.

[51] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke, "Security for grid services," in *Proc. 12th IEEE International Symposium on High Performance Distributed Computing*, pp. 48–57, 2003.

[52] A. Ferrari, F. Knabe, M. Humphrey, S. J. Chapin, and A. S. Grimshaw, "A flexible security system for metacomputing environments," in *Proc. the 7th International Conference on High-Performance Computing and Networking*, (London, UK), pp. 370–380, Springer-Verlag, 1999.

[53] A. O. Freier, P. Karlton, and P. C. Kocher, "The ssl protocol version 3.0." Internet Draft, January 1996.

[54] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol." RFC 4346, April 2006.

[55] "Openssl: The open source toolkit for ssl/tls." URL: http://www.openssl.org.

[56] "Ws-security specification." URL: http://www.oasis-open.org/specs/index.php#wssv1.0.

[57] "Web services secure conversation language." URL: http://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf.

[58] "Web services security policy language." URL: http://www6.software.ibm.com/software/developer/library/ws-secpol.pdf.

[59] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, "Wide area computing: resource sharing on a large scale," *Computer*, vol. 32, no. 5, pp. 29–37, 1999.

[60] M. Humphrey, "From legion to legion-g to ogsi.net: Object-based computing for grids," in *Proc. the 17th International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 2003.

[61] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-g: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, pp. 237–246, July 2002.

[62] N. Peyrouze and G. Muller, "Ft-nfs: an efficient fault-tolerant nfs server designed for off-the-shelf workstations," in *Proc. the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, (Washington, DC, USA), IEEE Computer Society, 1996.

[63] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. Symposium on Operating Systems Design and Implementation*, USENIX Association, 1999.

[64] A. I. T. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *Proc. Symposium on Operating Systems Principles*, pp. 188–201, 2001.

[65] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. 5th Symposium on Operating Systems Design and Implementation*, 2002.

[66] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe, "Wide area data replication for scientific collaborations," in *Proc. The 6th IEEE/ACM International Workshop on Grid Computing*, pp. 1–8, 2005.

[67] The, "Web services architecture." public draft, http://www.w3.org/TR/2003/WD-ws-arch-20030808/, August 2003.

[68] K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana, "Modeling stateful resources using web services," Globus Alliance, March 2004.

[69] "Globus toolkit." URL: http://www.globus.org/toolkit/.

[70] G. Wasson and M. Humphrey, "Exploiting wsrf and wsrf.net for remote job execution in grid environments," in *Proc. the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, (Washington, DC, USA), IEEE Computer Society, 2005.

[71] "Wsrf::lite." URL: http://www.sve.man.ac.uk/Research/AtoZ/ILCT.

[72] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[73] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *Proc. International Conference on Autonomic Computing*, pp. 2–9, 2004.

[74] T. Kosar, G. Kola, and M. Livny, "A framework for self-optimizing, fault-tolerant, high performance bulk data transfers in a heterogeneous grid environment," in *Proc. Second International Symposium on Parallel and Distributed Computing*, pp. 137–144, 2003.

[75] V. Kalogeraki, "Decentralized resource management for real-time object-oriented dependable systems," tech. rep., 2001.

[76] D. Murky, C. David, W. Ian, S. Alla, G. Pawan, S. Aamer, R. Keri, L. Ed, T. William, A. Bill, B. Marcus, and K. Alexander, "Policy-based autonomic storage allocation," in *Proc. IFIP/IEEE International workshop on Distributed systems, Operations and Management*, October 2003.

[77] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen, "Globedb: autonomic data replication for web applications," in *Proc. the 14th international conference on World Wide Web*, (New York, NY, USA), pp. 33–42, ACM Press, 2005.

[78] H. Yu and A. Vahdat, "Consistent and automatic replica regeneration," *Trans. Storage*, vol. 1, pp. 3–37, February 2005.

[79] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 3–12, July 2003.

[80] G. Venkitachalam and B. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *USENIX Annual Technical Conference*, 2001.

[81] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proc. the ACM Symposium on Operating Systems Principles*, October 2003.

[82] J. Dike, "A user-mode port of the linux kernel," in *Proc. 4th Annual Linux Showcase and Conference*, 2000.

[83] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," in *Proc. the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[84] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The collective: A cache-based system management architecture," in *Proc. the 2nd Symposium on Networked Systems Design and Implementation*, pp. 259–272, May 2005.

[85] C. Clark, K. Fraser, and H. Steven, "Live migration of virtual machines," in *Proc. the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–11, 2005.

[86] "Samba." URL: http://us4.samba.org/samba/.

[87] R. J. Figueiredo, N. Kapadia, and J. A. B. Fortes, "Seamless access to decentralized storage services in computational grids via a virtual file system," *Cluster Computing*, vol. 7, pp. 113–122, April 2004.

[88] "Iozone file system benchmark." URL: http://www.iozone.org.

[89] "Iperf: The tcp/udp bandwidth measurement tool." URL: http://dast.nlanr.net/Projects/Iperf/.

[90] J. Katcher, "Postmark: A new file system benehmark," tech. rep., Network Appliance, 1997.

[91] J. Paladugula, M. Zhao, and R. J. Figueiredo, "Support for data-intensive, variable-granularity grid applications via distributed file system virtualization — a case study of light scattering spectroscopy," in *Proc. the Second International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 2004)*, pp. 12–21, 2004.

[92] J. Hennessy and D. Patterson, *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 3rd edition ed., 2002.

[93] M. Zhao and R. J. Figueiredo, "Proxy managed client-side disk caching for the virtual file system," tech. rep., November 2004.

[94] M. Zhao, V. Chadha, and R. J. Figueiredo, "Supporting application-tailored grid file system sessions with wsrf-based services," in *Proc. 14th IEEE International Symposium on High Performance Distributed Computing(HPDC-14)*, pp. 24–33, 2005.

[95] M. Carson and D. Santay, "Nist net: a linux-based network emulation tool," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 111–126, July 2003.

[96] "Ti-prc." URL: http://nfsv4.bullopensource.org/doc/tirpc_rpcbind.php.

[97] A. Zeitoun, Z. Wang, and S. Jamin, "Rttometer: measuring path minimum rtt with confidence," in *Proc. 3rd IEEE Workshop on IP Operations and Management (IPOM 2003).*, pp. 127–134, 2003.

[98] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1998.

[99] "The keyed hash message authentication code (hmac)." FIPS Pub 198, 2002.

[100] K. Kaukonen and R. Thayer, "A stream cipher encryption algorithm 'arcfour'." Internet Draft, 1999.

[101] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design.* Addison-Wesley, 3rd edition ed., 2001.

[102] "Sura coastal ocean observing and prediction (scoop) program." URL: http://scoop.sura.org.

[103] A. Butt, S. Adabala, N. Kapadia, R. Figueiredo, and J. Fortes, "Grid-computing portals and security issues," *Journal of Parallel and Distributed Computing*, vol. 63, no. 10, pp. 1006–1014, 2003.

[104] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer Magazine*, vol. 7, no. 6, pp. 34–45, 1974.

[105] "Gsi-enabled openssh." URL: http://grid.ncsa.uiuc.edu/ssh/.

[106] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, "Vmplants: Providing and managing virtual machine execution environments for grid computing," in *Proc. the 2004 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), IEEE Computer Society, 2004.

[107] VMware Inc., *VMware VirtualCenter Users Manual.*

[108] VMware Inc., *GSX Server 2.5.1 User's Manual.*

[109] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration." Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[110] H. Kreger, "Web services conceptual architecture," IBM Software Group, 2001.

[111] N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes, "Enhancing the scalability and usability of computational grids via logical user accounts and virtual file systems," in *Proc. 10th Heterogeneous Computing Workshop*, pp. 82–82.

[112] S. Adabala, A. Matsunaga, M. Tsugawa, R. Figueiredo, and J. A. B. Fortes, "Single sign-on in in-vigo: role-based access via delegation mechanisms using short-lived user identities," in *Proc. 18th International Parallel and Distributed Processing Symposium*, pp. 26–30, April 2004.

[113] V. Chadha and R. J. Figueiredo, "Row-fs: A user-level virtualized redirect-on-write distributed file system for wide area applications," in *Proc. 13th Int. Conf. HiPC*, 2007.

[114] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of unix processes in the condor distributed processing system," tech. rep., Univ. of Wisconsin-Madison, 1997.

[115] M. Bozyigit and M. Wasiq, "User-level process checkpoint and restore for migration," *ACM SIGOPS Operating Systems Review*, vol. 35, pp. 86–96, April 2001.

[116] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke, "A community authorization service for group collaboration," in *Policies for Distributed Systems and Networks, 2002. Proc. Third International Workshop on*, pp. 50–59, 2002.

[117] O. Sato, R. Potter, M. Yamamoto, and M. Hagiya, "Uml scrapbook and realization of snapshot programming environment," in *Proc. the International Symposium on Software Security*, 2003.

[118] J. Xu, S. Adabala, and J. A. B. Fortes, "Towards autonomic virtual applications in the in-vigo system," in *Proc. Second International Conference on Autonomic Computing (ICAC 2005)*, pp. 15–26, 2005.

[119] Y. Zhong, S. Dropsho, and C. Ding, "Miss rate prediction across all program inputs," in *Proc. 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*, 2003.

[120] R. Wolski, "Dynamically forecasting network performance using the network weather service," *Cluster Computing*, vol. 1, no. 1, pp. 119–132, 1998.

[121] E. J. Gardner, "Exponential smoothing: The state of the art–part ii," *International Journal of Forecasting*, vol. 22, no. 4, pp. 637–666, 2006.

[122] D. C. Anderson, J. S. Chase, and A. M. Vahdat, "Interposed request routing for scalable network storage," *ACM Trans. Comput. Syst.*, vol. 20, no. 1, pp. 25–48, 2002.

[123] L. W. Russell, S. P. Morgan, and E. G. Chron, "Clockwork: A new movement in autonomic systems," *IBM Systems Journal*, vol. 42, no. 1, 2003.

[124] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante, "Modeling and taming parallel tcp on the wide area network," in *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[125] "nanohub." URL: http://www.nanohub.org/.

[126] W. C. Feng and P. Tinnakornsrisuphap, "The failure of tcp in high-performance computational grids," in *Proc. High-Performance Networking and Computing Conf.*, 2000.

[127] H. R. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (H. Jin, T. Cortes, and R. Buyya, eds.), pp. 224–244, New York, NY: IEEE Computer Society Press and Wiley, 2001.

BIOGRAPHICAL SKETCH

Ming Zhao was born in Shanghai and grew up in Wuhan, two beautiful cities along the Yangzi River in China. At the age of nineteen, he left his hometown in pursuit of higher education, which has continued till today. Through seven years of study at Tsinghua University, Beijing, China, he received the B.E. and M.E. degrees from the Department of Automation, with a specialty in pattern recognition and intelligent systems. During the course of his master's dissertation work, a Web-based system for content-based image retrieval, he developed strong interests in computer systems research. Encouraged by his parents and advisor, he decided to continue his study overseas, in the United States of America.

In 2001, he started as a Ph.D. student at Rice University, Houston, TX. However, it was not until 2003, when he followed Prof. Renato Figueiredo to join Prof. José Fortes' ACIS lab at the University of Florida, Gainesville, FL, that he finally found the research he is fascinated with, distributed systems and virtualization. During more than five years of research in this area, he has actively published in the related conferences and journals, and the systems he built have also been deployed for production use, servicing applications and users from many disciplines. It has been quite a long journey for him in this intellectual quest, and he will continue it as an Assistant Professor at the School of Computing and Information Sciences at Florida International University, Miami, FL, in Fall 2008.