User-Level Remote Data Access in Overlay Metacomputers

Jeff Siegel and Paul Lu

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{siegel|paullu}@cs.ualberta.ca

Abstract

A practical problem faced by users of metacomputers and computational grids is: If my computation can move from one system to another, how can I ensure that my data will still be available to my computation? Depending on the level of software, technical, and administrative support available, a data grid or a distributed file system would be reasonable solutions. However, it is not always possible (or practical) to have a diverse group of systems administrators agree to adopt a common infrastructure to support remote data access. Yet, having transparent access to any remote data is an important, practical capability.

We have developed the Trellis File System (Trellis FS) to allow programs to access data files on any file system and on any host on a network that can be named by a Secure Copy Locator (SCL) or a Uniform Resource Locator (URL). Without requiring any new protocols or infrastructure, Trellis can be used on practically any POSIX-based system on the Internet. Read access, write access, sparse access, local caching of data, prefetching, and authentication are supported. Trellis is implemented as a user-level C library, which mimics the standard stream I/O functions, and is highly portable. Trellis is not *a replacement for traditional file systems or data grids; it provides new capabilities by overlaying on top of other file systems, including grid-based file systems. And, by building upon an already-existing infrastructure (i.e., Secure Shell and Secure Copy), Trellis can be used in situations where a suitable data grid or distributed file system does not yet exist.*

Keywords: *remote data access, wide area file systems, user-level file system, metacomputing, overlay metacomputers, grid computing, prefetching, caching*

1 Introduction

High-speed wide-area networks (WAN) make it more attractive to take advantage of computational resources at different computing centers. But, in practice, users tend to access only the computers at their local center because that is where their data is located. For metacomputing and grid computing to flourish, applications must be able to run on any computer at any site and still have transparent access to their data files.

Traditional distributed file systems allow remote volumes to be accessed locally. A disk volume appears to be local, but it is actually accessed in client-server fashion from the remote file server. For example, the Network File System (NFS) [11] and the Andrew File System (AFS) [7] are distributed file systems that have been used productively for many years. Both NFS and AFS (and similar systems) can be viewed as formal file systems in the sense that they require a superuser to install the appropriate drivers or modules into the operating system (OS), require a superuser to configure mount points, and provide all users with a transparent view of a whole file system, regardless of whether a volume is local or remote. In general, an unprivileged user cannot create, mount, or reconfigure any aspect of the formal file system. Nonetheless, traditional distributed file systems are powerful and useful systems.

But, currently, many researchers have access to a variety of different computer systems that do not share a data grid or distributed file system (Figure 1). The researcher merely has an account on the systems. For example, Researcher A has access to their group's system, a departmental system, and a system at a high-performance computing center. Researcher B has access to their group's server and (perhaps) a couple of different high-performance computing centers, including one center in common with Researcher A. It would be ideal if all of the systems could be part of one metacom-

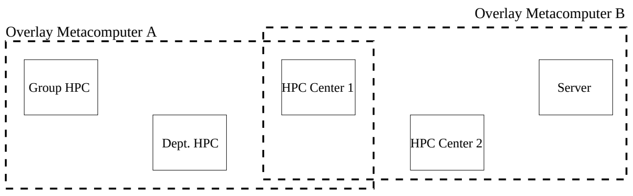

Figure 1. Overlay Metacomputers

puter or computational grid. But, practically, the different systems may be controlled by different groups who may not run the same grid software. Yet, Researchers A and B would still like to be able to exploit the large aggregate power of their systems.

One solution is to create an *overlay metacomputer*, which is a user-level aggregate of individual computing systems (Figure 1). A practical and usable overlay metacomputer can be created by building upon existing networking and software infrastructure, such as Secure Shell, Secure Copy, and World Wide Web (WWW) protocols. Since the infrastructure is entirely at the user-level, or part of a well-supported, existing infrastructure, Researcher A can create a personal Overlay Metacomputer A. Similarly, Researcher B can create a personal Overlay Metacomputer B, which can overlap with Researcher A's metacomputer (or not). Our strategy for automatically handling computational tasks in an overlay metacomputer is described elsewhere [9].

We have developed the Trellis File System (Trellis FS) to help address the remote data access problem. Since Trellis is designed to work on overlay metacomputers, we can describe Trellis as an *overlay file system*. An important design decision in Trellis is to, as much as possible, implement all the basic functionality at the user-level. Consequently, Trellis provides a rich set of features, including read access, write access, sparse access, local caching of data, prefetching, and a flexible approach to authentication and security.

2 Trellis File System

File systems provide a convenient abstraction to access data files organized in a namespace. Traditional local file systems abstract the details of how disk blocks are allocated and mapped to a (typically) hierarchical namespace. Distributed file systems abstract the details of client-server access of remote data stored in various local file systems. A superuser is required to configure most local and distributed file systems. One can layer other systems for linking and accessing data across a network on top of existing file systems. For example, the WWW can be viewed as a (largely) read-only file system where a Uniform Resource Locator (URL) is the basic element of the namespace. Notably, it is possible for users to set up personal Web servers without

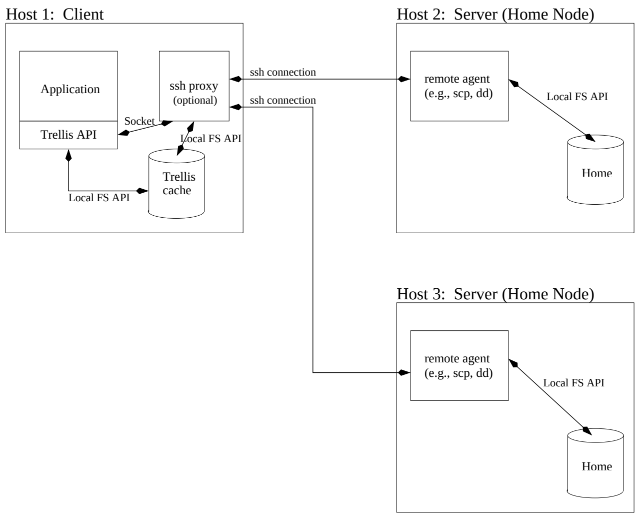

Figure 2. Trellis Architecture

being a superuser.

Trellis is an application programmer's interface (API) and system for transparently accessing local and remote data. Any data file that can be named and accessed via a URL can also be accessed via the Trellis File System (Trellis FS). In addition, Trellis can access data files that are reachable via Secure Shell (`ssh`) and Secure Copy (`scp`). The naming convention for `scp` is similar to a URL, with a few syntactic differences. We refer to the Secure Copy naming syntax of `username@hostname:pathname` as a Secure Copy Locator (SCL). An important advantage of `scp` is the built-in infrastructure for user authentication via public key encryption instead of passwords. Most Web servers provide simple forms of authentication (e.g., Apache's `htaccess`) based on passwords. By using public key encryption, access to data files can be given and revoked without revealing any private information. In addition, `ssh` allows more fine-grained approaches to access control via the *forced command* feature [3]. Given the wide deployment of Secure Shell and Copy, we have layered Trellis on top of that existing infrastructure.

To the application programmer, the Trellis FS can be viewed as a library of wrapper functions to well-known C language stream I/O functions (Table 1). For example, instead of `open()`, the programmer calls `trellis_open()` with the same parameter types. Both functions return a file descriptor.

Trellis transparently replicates a remote file onto a local file system and then allows the data to be accessed using `trellis_read()` or `trellis_write()`. Once cached on the local file system, usually in a directory called `TrellisCache`, Trellis delegates to the local file system the actual movement of data from the file cache to the user's

Function Name	Description
`trellis_open()`	Opens a local file, remote file via URL, or remote file via Secure Copy. Same parameters as `open()`. Currently, flags `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, and `O_APPEND` are supported.
`trellis_close()`	Closes any file opened by `trellis_open()`. Flushes updates to remote host. Same parameters as `close()`.
`trellis_read()`	Reads from a file opened by `trellis_open()`. Data comes from a copy of the remote file cached on the local file system. Same parameters as `read()`.
`trellis_write()`	Writes to a file opened by `trellis_open()`. Updated values are buffered on a local file system until file is closed. Same parameters as `write()`.
`trellis_lseek()`	Moves current file location in a file opened by `trellis_open()`. Supports sparse files. Same parameters as `lseek()`.

Table 1. Main Functions in Trellis File System API

buffers. Therefore, Trellis is not responsible for disk block allocation or other low-level data management details. In the common use cases, Trellis introduces only a small function call overhead to each I/O operation. To improve performance, data can be prefetched into the Trellis cache. And, the cache is persistent so the remote data transfer can be eliminated for future accesses to the same data file.

Accessing files via an SCL is the most general-purpose mechanism since `scp`/`ssh` can both read and write files. When `trellis_close()` is called, any updates to the file are written back to the original host (called the *home node*) if `scp`/`ssh` was the original protocol.

We converted the well-known Unix `cat` program to Trellis by making only a few lexical changes to the source code to use the `trellis_` functions. The new `trellis_cat` accepts the same command-line arguments and options as `cat`, but it also accepts the URL and SCL filenames. Consequently, `trellis_cat` can be used in scripts as a drop-in replacement for `cat`.

2.1 Overlay File Systems

We have described Trellis as an *overlay file system*. The term "overlay" is borrowed from the notion of overlay networks [2]. We define the three main characteristics of an overlay file system as:

1. *Does not require special kernel support.* Although an overlay file system may benefit from special kernel support, an overlay file system cannot require such support because that could severely limit portability.

2. *Does not require superuser permissions to set up or use.* The current Trellis prototype does not require mount points and all configuration is controlled via shell environment variables.

3. *Does not require all users to see the same data file namespace; users* **can** *share the same namespace if they wish.* Different users have access to different hosts and computational nodes. Compare Researcher A with B (Figure 1). Therefore, there is unlikely to be a single directory structure that will satisfy all users.

3 Implementation of Trellis FS

The Trellis FS architecture is shown in Figure 2. The client application is linked to the Trellis FS user-level C library, which implements the API. For our prototype implementation of the Trellis FS, we made a deliberate decision to re-use as much existing technology as possible. For example, instead of designing a new scheme for user authentication and data encryption, we adopted the Secure Shell infrastructure. Also, `libcURL` [8] was already a robust library that supported a wide variety of URL-related data transfer protocols. Consequently, except for `scp`-accessed files, we call the appropriate `libcURL` functions to transfer the remote file. On top of `libcURL` and Secure Shell, Trellis adds data management functionality, the Trellis cache, simple prefetching, and sparse file support.

As discussed earlier, we tested the abstraction benefits of the Trellis FS API by converting the `cat` program into `trellis_cat`. Since the Trellis functions are designed to be drop-in replacements for the standard stream I/O functions, porting `cat` to use the Trellis FS required only a handful of lexical changes to the source code.

Microbenchmarks (not discussed here) on Linux 2.2 and Irix 6.5 systems show that Trellis has a small performance penalty of 5% to 10% over explicit file copying and using the original binary file API. The Trellis overheads include data management of the Trellis cache and (most importantly) the function call redirection overheads of using the `trellis_` functions. However, when prefetching and caching are allowed and exploited in Trellis, there can be substantial performance gains. For computationally-intensive applications, both prefetching and caching can reduce execution times. For simple computations, caching is the most effective technique. Lastly, for applications with sparse data access patterns, Trellis's ability to transfer files on demand in blocks can result in large performance improvements.

Further implementation and performance evaluation details can be attained by contacting the authors.

4 Related Work

We have already discussed the relationship between overlay file systems and traditional, distributed file systems, such as NFS [11] and AFS [7].

Of course, the basic idea of using wrapper functions (e.g., `trellis_open()` instead of `open()`) is not new. Orthogonally, we are making progress towards more transparent interfaces than wrapper functions, in order to support unmodified binary applications.

Trellis also draws heavily on the concept of a URL from the WWW and on WWW-related systems, such as `libcURL` [8] and Ufo [1].

The influential work of the Globus project on GASS and the Data Grid is related to Trellis [6, 4, 10]. Also, there are other grid-related remote data access systems [12, 5]. The systems strive to make it easier to access remote data in a convenient manner and with good performance.

5 Concluding Remarks

The Trellis File System provides an abstraction to allow for the transparent access of remote data files through URL-based and SCL-based filenames. In the rapidly emerging areas of metacomputing and grid computing, having transparent access to any remote data is an important capability.

Trellis is unique in its design philosophy (i.e., overlay file systems) and its use of Secure Shell and Copy as a fundamental building block. Read access, write access, sparse access, local caching of data, prefetching, and strong user authentication are supported. Microbenchmark experiments show the relatively low overheads introduced by Trellis and the potential performance advantages of prefetching and caching in Trellis. Few remote data access systems for metacompting provide all of these features in a system that is ready to deploy with the existing network infrastructure.

Finally, as an overlay file system, Trellis can be installed, configured and re-configured without requiring superuser access. Trellis is *not* a replacement for traditional file systems or data grids; it provides new capabilities by building upon other file systems and by layering on top of widely-deployed network infrastructure. In the future, Trellis applications can happily co-exist on a variety of grid infrastructures and, as the project progresses, Trellis can provide additional functionality for users of metacomputers and grids.

Acknowledgments

Thank you to C3.ca, the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada Foundation for Innovation (CFI), and the University of Alberta for their research support.

References

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems*, 16(3):207–233, 1998.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In G. Ganger, editor, *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 131–145, New York, Oct. 21–24 2001. ACM Press.

[3] D. J. Barrett and R. E. Silverman. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly and Associates, Sebastopol, CA, 2001.

[4] 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. 6th Workshop on I/O in Parallel and Distributed Systems*, 1999.

[5] R. Figueiredo, N. Kapadia, and J. Fortes. The PUNCH virtual file system: Seamless access to decentralized storage services in a computational grid. In *Proc. 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[6] Globus. `http://www.globus.org/`.

[7] J. H. Howard. On overview of the Andrew File System. In USENIX Association, editor, *USENIX Conference Proceedings (Dallas, TX, USA)*, pages 213–216, Berkeley, CA, USA, Winter 1988. USENIX.

[8] libcURL. `http://curl.haxx.se/libcurl/`.

[9] C. Pinchak, P. Lu, and M. Goldenberg. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, UK, July 2002.

[10] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and Object Replication in Data Grids. In *Proc. 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[11] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In USENIX Association, editor, *Proceedings: USENIX Association Winter Conference, January 23–25, 1985, Dallas, Texas, USA*, pages 117–124. USENIX, Winter 1985.

[12] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In ACM, editor, *Supercomputing 2001*. ACM Press and IEEE Computer Society Press, 2001.