University of Alberta

THE TRELLIS NETWORK FILE SYSTEM

by

Michael Closson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95725-X
Our file Notre référence
ISBN: 0-612-95725-X

To Taunia

Acknowledgements

I would like to acknowledge my supervisor Dr. Paul Lu for all his guidance, inspiration and patience. TrellisNFS would not be possible without the efforts Jeff Siegel and Paul Lu and the rest of the TrellisFS team: Nolan Bard, Morgan Kan and Mark Lee. TrellisNFS would also not have been possible without the efforts of Morgan Kan and Danny Ngo who implemented the Trellis Security Infrastructure. I also wish to acknowledge the work of additional members of the Trellis team: Meng Ding, Mark Goldenberg, Nick Lamb, Chris Pinchak, Dr. Jonathan Schaeffer, Ron Senda, Dr. Edmund Sumbar and Yang Wang.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a Trellis metacomputer [21], compute jobs are performed on a geographically distant virtual computer composed from multiple independent computing resources. Compute jobs are distributed across the metacomputer in such a way that any compute server is a candidate to perform any computing job [20]. A natural consequence of this fact is that the data required for a compute job must be made available to the compute server to which the job has been assigned [25].

Issues relating to data movement inside a Trellis metacomputer include:

- Where is the input data coming from and to where should the output data go?
- Since a Trellis metacomputer spans multiple administrative domains, do the respective servers have the necessary credentials to access each other's services?
- What are the primitives available for accessing and manipulating data?
- How is data located and named?
- What semantics are in place to deal with multiple copies of data? How and when are the multiple copies kept synchronized?
- Issues that have perhaps not received much consideration by previous distributed file systems are the relevant social concerns. For example, since a Trellis metacomputer spans multiple administrative domains, what burden does a distributed file system place on the respective system administrators in terms of installation and use?

We present the Trellis Network File System (TrellisNFS); which allows the various computing resources in a Trellis metacomputer to access and manipulate data on the metacomputer's various data storage resources. TrellisNFS accomplishes this by integrating the benefits of the Network File System [23] (NFS) and the Trellis File System [25] (TrellisFS).

NFS has some features that make it useful as a metacomputing file system: it supports all the file system functionality associated with a traditional Unix file system, provides primitives for file

access that can be used by unmodified binaries, and uses a hierarchal namespace. On the other hand, there are some features of NFS that make it unsuitable to be used as a metacomputing file system. NFS was designed to run on a local area network (LAN). NFS features, such as synchronous operations are designed to be used in a low latency network. A high latency network such as the wide area networks (WANs) of the Internet would adversely impact the performance of NFS. NFS was designed to work in a single administrative domain, and the security model of NFS makes the assumption that it will be used in a tightly controlled environment. Also, installing NFS is the job of a system administrator, and cannot be performed by ordinary users.

Likewise; the TrellisFS library, which is a component of TrellisNFS, has some advantages that make it useful as a metacomputing file system. TrellisFS uses strategies to offset high WAN latencies and has a security model that allows it to work across multiple administrative domains. TrellisFS is completely user-installable and does not require system administrator support. However, the disadvantage of the TrellisFS library is that it exposes an Application Programming Interface (API) as the primitive for client integration; therefore, applications that wish to take advantage of the TrellisFS library must be modified at the source code level and re-compiled. In the case of commercial software, the source code may not be available; or getting access to the source code may be possible only by purchasing an additional licence.

The goal of the TrellisNFS server is to leverage the benefits of these two file systems. Explicitly, the goals of TrellisNFS are to:

- Allow file sharing over a WAN;
- Work across multiple administrative domains;
- Seamlessly integrate with the existing local file system;
- As much as possible, remain user-installable;
- Maintain the security benefits provided by TrellisFS;
- Prevent undermining of a system administrator's ability to control the security of her own system.

1.1 The Trellis Project

We have designed TrellisNFS to be a component of the Trellis metacomputing system. A Trellis metacomputer is a virtual, batch-processing, capacity-oriented computer; comparable to a computing cluster. The services provided by a Trellis metacomputer are similar to those provided by a computing cluster; we list these similar services now:

- A computing cluster provides a batch-scheduler such as the Portable Batch System (PBS), to allocate CPU time to compute jobs. The Trellis metacomputer provides a batch-scheduler [20] that is based on placeholder scheduling.

- Computing clusters are part of a single administrative domain. Existing tools for user authentication and authorization are provided. As part of the Trellis project, a security infrastructure [17] has been developed for use in a Trellis metacomputer.

- Computing clusters use a shared file system for managing data. TrellisNFS provides an integrated distributed file system for a Trellis metacomputer.

1.2 A Motivating Example

Many areas of research in the natural sciences rely on computers to assist in their understanding of natural phenomena. Researchers develop complex numerical models that require large amounts of computational power to simulate; simulation is a powerful technique to aid in the design and verification of these numerical models of natural phenomena.

As a concrete example, we will use the Gromacs [18, 6] molecular dynamics simulator. Gromacs is a CPU-bound sequential application. Multiple runs of different molecular configurations can be executed in parallel. Input files are typically a few megabytes in size, and the size of the output data is about 10 times that of the input data. These characteristics are typical of High Performance Computing (HPC) application programs.

If a researcher has access to multiple HPC resources (e.g., multiple compute clusters) that are in different administrative domains, then distributing the simulation across these resources is more difficult then confining the simulation to a single cluster. For example, multiple HPC resources do not have a common batch scheduler or a common file system. It would be of great benefit if using these multiple HPC resources was as simple and convenient as using a single HPC resource.

This problem is common in computational science, and for this purpose the Trellis project was initiated. In this work we focus particularly on a shared file system for a Trellis metacomputer. By deploying a TrellisNFS server in each administrative domain, a researcher can benefit from the power and convenience of the shared file system of a single HPC resource, while using multiple HPC resources in multiple administrative domains.

1.3 Contributions

The goal of metacomputing is to harness the collective computing power of existing HPC consortia. A shared file system in a computing cluster is powerful because if allows users to access files from across the network by using existing file system primitives. Additionally, a shared file system allows

the user to centrally locate their data and still have it be transparently accessible by all nodes in the cluster. The power of a shared file system in a computing cluster has not been fully realized in a Trellis metacomputer, for this purpose we have designed and implemented the Trellis Network File System. This work makes three contributions:

1. We have re-designed and re-implemented the Linux UNFSD server [24], allowing it to work with remote files; the TrellisNFS server is based on this server.

2. We have expanded the functionality of the TrellisFS library. The original version of the TrellisFS library allowed a client only to access and manipulate remote file data. In order to integrate the TrellisFS library with the user-level NFS server; we expanded the scope and functionality of the library to allow it to work with directories, metadata, hard and symbolic links, and file renaming. In addition, we implemented the TrellisFS metadata cache, a general purpose mechanism to eliminate redundant metadata queries in TrellisFS clients.

3. We designed and implemented a Remote Procedure Call (RPC) over SSH [30] mechanism. This mechanism was built on top of the SSH Proxy [25]. The SSH Proxy is a framework for persistent SSH connections.

1.4 Concluding Remarks

We have discussed our motivation for building the TrellisNFS server and have given a motivating example of how we envision using it. In the next chapter, we will discuss the issues of distributed file systems and look at some of the related work in the field.

Chapter 2

Background and Related Work

In the previous chapter we introduced the Trellis Network File System and discussed the motivation for implementing the TrellisNFS server. We now present important concepts applicable to the design and implementation of a distributed file system. We also outline previous work in the field.

2.1 Background concepts

Before we present the architecture of the TrellisNFS server in the next chapter, there are some important concepts to introduce first, which will facilitate later discussion.

The background concepts we discuss in this section can be grouped into seven categories: 1) We characterize the difference between a distributed data storage system and a distributed file system; 2) We discuss file system operations common to a typical high performance computing workload; 3) We introduce important concepts relating to the Secure Shell [30]. The Secure Shell is used extensively by the TrellisNFS system for authorization, authentication and data encryption; 4) We introduce Secure Copy Locater (SCL) notation [25], the notation we use for referring to files in the Trellis namespace; 5) We describe fundamental concepts that relate to all classes of file systems; 6) We discuss concepts relating specifically to NFS; 7) Finally, we present concepts relating specifically to the TrellisNFS server.

2.1.1 Distributed Data Storage Systems versus Distributed File Systems

Systems for accessing remote data can be grouped into two categories. One category, *Distributed Data Storage Systems*, includes common tools for retrieving data on a remote system. Examples include the File Transfer Protocol (FTP) and the Secure Shells' `scp` utility. The other category, *distributed file systems*, includes examples such as the Network File System (NFS) or Coda.

Distributed data storage systems and distributed file systems both have the basic capability for accessing data on a remote server, but distributed file systems provide full file system semantics. File system semantics include features such as the following:

1. *Distributed file systems implement a full file system API*: A distributed file system provides functionality to access and manipulate files, directories, hard and symbolic links, and metadata. In contrast, a distributed data storage system will only support replication of data. Most distributed data storage systems only provide functionality to transport data between a remote node and the local node.

2. *Distributed file systems provide defined file system consistency guarantees*: File system consistency deals with the notion of how the contents of multiple copies of the same file are kept synchronized with each other. If updates are propagated after every write, and are therefore immediately visible to all potential readers, we say that the file system has strong consistency guarantees. If file updates are propagated infrequently, we say the file system has weak consistency guarantees. There is a trade-off between strong and weak file system consistency. Strong consistency generally leads to a slower file system because file operations will have to communicate more frequently with other nodes to determine if cached data has changed. Also, maintaining strong consistency means potentially wasting network bandwidth. On the other hand, weak consistency means that all readers may not have the most recent data and that the file system may not correctly handle multiple clients writing to the same file at the same time. The decision about what strength of file system consistency to support depends on the nature of the workloads that applications will place on the file system. It is generally accepted that no single file system consistency policy is optimal for all file systems.

   There are 3 common file consistency policies: 1) *write-to-read* consistency, 2) *close-to-open* consistency, and 3) *last-writer-wins* consistency. This is not an exhaustive list of the different levels or kinds of file system consistency, but is meant to illustrate the spectrum of possible options. Write-to-read consistency is the strongest of the three; it means that if any process reads from a file it will see all past writes performed by any writer. That is, after a file system write by any writer, the next read performed by any reader will see the most up-to-date file data. Write-to-read consistency is the consistency policy associated with a local disk file system. Close-to-open consistency, also referred to as session consistency, is weaker then write-to-read consistency. Close-to-open consistency, as the name suggests, means that when a process opens a file, it will see all changes from all past writers who have already closed the file. This means that if two writers both have a file open at the same time, then changes to the file made by one of these writers may be lost. Even weaker than close-to-open consistency is last-writer-wins consistency. As the name suggests, the last process to write to the file will potentially overwrite changes made by any previous writer. It should be noted that the order of writes is not necessarily determined by the time at which a write function call completes, but most likely at the time the master copy of the file is updated.

A distributed file system provides more guarantees when dealing with multiple readers and multiple writers then do remote data storage systems, which typically only support last-writer-wins semantics. When dealing with multiple copies of a file, a remote data storage system may not even be able to provide last-writer-wins semantics.

3. *Distributed file systems support seamless and transparent integration with the existing file system hierarchy*: A distributed file system integrates with the file system hierarchy of the local machine to provide seamless access to remote files. Specifically, a distributed file system allows an application to use the existing file access primitives provided by the operating system, rather then implement new primitives. For example, the `open()` function in a distributed file system will return a file descriptor, and this file descriptor can be operated on through existing operating system functions. In contrast, a distributed data storage system will provide primitives that are incompatible with those provided by the operating system.

This is not meant to be an exhaustive list, but it is meant to illustrate some of the reasons that have motivated us to design and implement the TrellisNFS server.

2.1.2 Typical File System Operations in a High Performance Workload

Different applications create different demands on the file system. TrellisNFS is designed to be useful for HPC. Sequential, whole-file access is the common access pattern of HPC application; operations such as accessing only a portion of a file, directory operations, and symbolic/hard link operations are rare. Although TrellisNFS supports all Unix file system operations, we have optimized the TrellisNFS server and the TrellisFS library for reads and writes.

2.1.3 The Secure Shell

TrellisNFS and TrellisFS use the Secure Shell [30, 5] (SSH) for authentication, authorization and data encryption. The SSH enables end-to-end privacy for data sent over an insecure network: all traffic between a home node and the TrellisNFS server is encrypted.

The SSH has several modes of authentication. The most common are host-based, public-key and password.

Host-based authentication is most commonly used inside a single administrative domain. If two hosts are declared to be equivalent, then a user on one host can connect to the account with the same user-id on the second host. A public-private key challenge-response system is used to allow the hosts authenticate to each other. Because of the security concerns associated with host-based authentication, it is often used only within a computing cluster that is not directly accessible from an external network.

With password authentication, as the name suggests, the user is required to enter a password to be authenticated; this password is encrypted before it is sent over the network. Password authentication is more secure then host-based authentication, but requires the user to interactively enter a password. This is not suitable for TrellisNFS, since entering a password is interactive in nature and the Trellis environment is a batch-processing environment.

Public-key authentication uses a challenge-response system with a public-private key pair to authenticate the user. The server has the public portion of the key pair, and uses this key to generate a challenge to the client. The client then decrypts the challenge with the private key and sends the response to the server. The server verifies this response to determine if access can be granted. Public key authentication is the most secure since no private information (such as a password or a private key) is sent over the network. For added security, private keys are stored on disk in an encrypted format; if the private key is in encrypted format, the user is required to enter a pass-phrase to decrypt the private key. This interactivity presents the same problem we identified above with password authentication. The problem is solved with the help of a standard SSH utility program, the *SSH agent*. The SSH agent runs as a daemon on the same machine as the TrellisNFS server, and the user can load private keys into the agent. When the server needs access to a remote server, it can use the private keys in the SSH agent to decrypt public key-based authentication challenges.

The current implementation of the TrellisNFS server uses a single SSH agent. The server is not selective about which key to use based on what user on the client initiated the request. Because of this, the current implementation of the TrellisNFS server is not suitable for use as a multi-user server.

2.1.4 Secure Copy Locater Notation: The Trellis Namespace

In this section, we introduce Secure Copy Locater (SCL) notation. We need a way to refer to files in the Trellis metacomputing system, and an SCL names a file in a Trellis metacomputer. SCL syntax mirrors the syntax used by the `scp` command, which is part of the SSH suite [30]. An SCL is analogous to a Uniform Resource Locater (URL) for the World Wide Web; it identifies a file or directory in a Trellis metacomputer. For example, the SCL

```
scp:closson@scovil.cs.ualberta.ca:dir1/file1
```

Refers to the file `file1` in directory `dir1`. Directory `dir1` is in user `closson`'s home directory. The file is located on the server `scovil.cs.ualberta.ca`. The user-id `closson` is used when authenticating with the server. `scp` refers to the Secure Copy via SSH access method.

Formally, an SCL has the following format.

```
<protocol>:[<username>@]<nodename>:[/]<location>
```

`protocol` can be one of `scp`, `http` or `ftp`; for the TrellisNFS server, we use only `scp`. `nodename` identifies the name of the Internet node where the file is located. `username` is the name of the account to use for authentication. If `username` is omitted, then the user name of the owner of the currently running process is used. Finally, `location` is the pathname of the file on the server. If the location is prefixed with a slash (`/`), then it is an absolute pathname. Without the slash, the location is relative to the user's home directory.

Integrating the TrellisFS library and NFS introduces a small inconsistency in SCL notation. In a Unix pathname, directories are delimited with a forward slash (`/`). In TrellisNFS, the nodename part of an SCL is considered to be a directory, and therefore should be properly delimited.

Consider the following SCL:

```
scp:closson@jasper-10:water.tpr
```

This SCL refers to the file `water.tpr` in the user's home directory on the node `jasper-10`. This presents an inconsistency with our TrellisNFS namespace. Assuming the TrellisNFS volume is mounted at `/trellis`, the TrellisNFS path:

```
/trellis/scp:closson@jasper-10:water.tpr
```

refers to a file under `closson`'s home directory, but `/trellis` is not `closson`'s home directory. This path is not legal in the current implementation of the TrellisNFS. TrellisNFS forces the user to use the entry `scp:closson@jasper-10:.`, when referring to SCLs that are *home directory relative* (i.e., paths that are relative to a home directory). So, the correct TrellisNFS syntax for this SCL would be:

```
/trellis/scp:closson@jasper-10:./water.tpr
```

SCLs with a `location` relative to the root of the remote file system hierarchy are not affected by the integration of TrellisFS with NFS. Below is an example of a *root relative* SCL:

```
/trellis/scp:closson@jasper-10:/scratch/closson/water.tpr
```

The file `water.tpr` is located in the directory `/scratch/closson` on server `jasper-10`. The user account `closson` will be used when authenticating to the server.

2.1.5 File System Concepts

In this section, we provide an introductory explanation of some fundamental Unix file system concepts. We first introduce the notion of a file system hierarchy and then introduce i-node numbers and device numbers.

A *File System Hierarchy* is the notion of a tree of files and directories. A typical Unix file system hierarchy is shown in Figure 2.1.

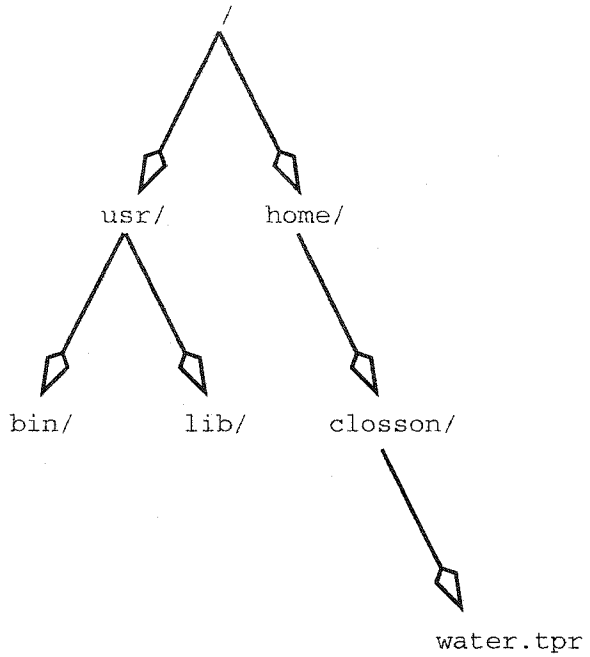


Figure 2.1: An illustration of a typical Unix file system hierarchy. Files and directories are organized in a tree structure. Non-leaf nodes are directories and leaf nodes can be either files or directories. Directories are suffixed with a `/`.

I-node numbers and device numbers are used extensively by NFS and by our implementation. In a Unix file system hierarchy, files are uniquely identified by their *device* and *i-node* numbers. Each file storage device, such as a partition on a hard drive, is assigned a unique device number by the operating system. On each file storage device, files are uniquely identified by an i-node number. It is possible for two files, each on a different device to have the same i-node number. In contrast, two files on the same Unix device *cannot* have the same i-node number. Also, Unix file system semantics are such that i-node and device numbers do not change during a file's or device's lifetime.

TrellisFS aims to make files from across the Internet available under a single device. Special care needs to be taken when assigning i-node numbers. It should be noted that most applications do not rely on these semantics for proper operation, i.e., most applications do not require that all files on a single Unix device have a unique i-node number; however, system software, such as the TrellisNFS server, does.

2.1.6 NFS Concepts

In this section we introduce the following concepts: NFS Volume, MOUNT Server, MOUNT Protocol, NFS Server, NFS Protocol; we define *mount*, and remote procedure calls (RPCs); we discuss the stateless nature of the NFS protocol, the NFS file handle, a NFS file-id number, the stateful nature of the MOUNT protocol, and the notion of synchronous operations.

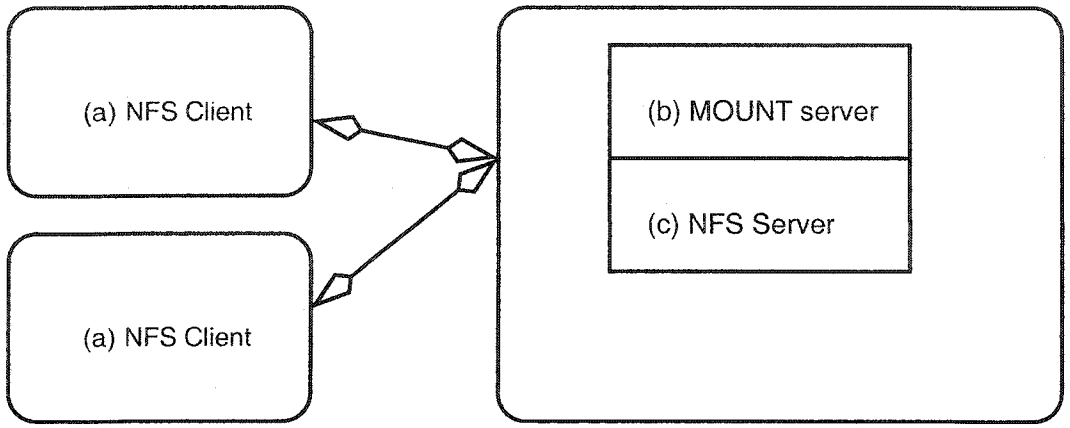


Figure 2.2: Architectural diagram of a typical NFS configuration. The main components are a) the client, b) the MOUNT server and c) the NFS server. NFS server accepts and processes requests from NFS clients.

NFS [23], is designed to allow file sharing over a LAN. An NFS server exports a hierarchy of files and directories to a client. A hierarchy like this is known as an *NFS volume*.

A typical NFS server setup is shown in Figure 2.2. an NFS server consists of two programs: the *MOUNT server* (object (b) in the Figure 2.2), which implements the *MOUNT protocol*, and the *NFS server* (object (c) in the figure 2.2), which implements the *NFS protocol*.

The MOUNT server is only used when establishing and dropping the mount; the NFS server does the majority of the work. The term *mount* refers to integrating a new file system hierarchy into an existing one.

A RPC is synonymous with a standard procedure call except that the process that invokes the procedure call and the process that executes the procedure code can be on different machines. A *synchronous* remote procedure call means that the process that calls the RPC will block until the remote machine has executed the procedure and returned the result; all RPCs in NFS are synchronous RPCs.

The NFS protocol is *stateless* by design. The server does not store any information for a specific client, which makes crash recovery simple. If the server restarts, it does not need to contact the clients to resynchronize their states. This stateless design means that a NFS server does not know when a process on a client opens or closes a file; all the server knows is that a file is being written to or read from. As we will discuss further in Sections 3.2.1 and 4.1.3, not knowing when a client application closes a file makes it difficult to know when to update the remote copy of a file with changes made to the cached copy (i.e., makes it difficult to implement *strong consistency*).

In NFS, the server and client use a data structure called a *NFS file handle* to identify files or directories. All NFS remote procedure calls use an NFS file handle as one of their arguments. NFS file handles are opaque data structures from the view of the NFS client. Before a client can perform

Remote Procedure Call	Description
NULL	Used for testing purposes.
GETATTR	Retrieve file or directory attributes.
SETATTR	Modify file or directory attributes.
LOOKUP	Retrieve NFS file handle.
READLINK	Retrieve the target of a symbolic link.
READ	Read from a file.
WRITE	Write to a file.
CREATE	Create a file.
MKDIR	Create a directory.
SYMLINK	Create a symbolic link.
REMOVE	Remove (delete) a file.
RMDIR	Remove (delete) a directory.
RENAME	Rename (move) a file.
LINK	Create a hard link.
READDIR	List directory contents.
STATFS	Retrieve file system information.

Table 2.1: A summary of NFS remote procedure calls.

any operation on a file, an NFS file handle must be obtained from the NFS server. The file handle referring to the root of the NFS volume is called the *root NFS file handle.*

For each file available in an NFS volume, the server generates an identification number unique to the file. This number is known as the *NFS file-id* number. Clients use this number as the Unix i-node number. We discuss NFS file-ids in Sections 3.2.1 and 4.1.3. Table 2.1 lists a summary of the 16 NFS procedure calls.

The MOUNT server performs two primary functions: it provides the root NFS file handle to the NFS client, and also maintains a list of which clients currently have an NFS volume mounted.

The MOUNT protocol is *stateful.* It tracks which clients have mounted which volume. Tracking state is the reason that the functionality of the MOUNT protocol is not integrated into the NFS protocol [10].

All NFS RPC operations are processed in the same request-response manner. The server receives the request, processes the request, and returns the result. For example, the client may request that data be written to a file. The server requires a file handle, an offset, and the data to be written, to perform the operation; the client provides these arguments to the server. The server will attempt the operation and return a result, which could be success or a failure code, such as "no space left" or "permission denied". The client blocks until the request is performed (refer to Figure 2.3 for an illustration of an NFS WRITE operation). This *synchronous* operation simplifies the design and implementation of an NFS server. Synchronous writes have proved to be a major obstacle to improving NFS performance [15].

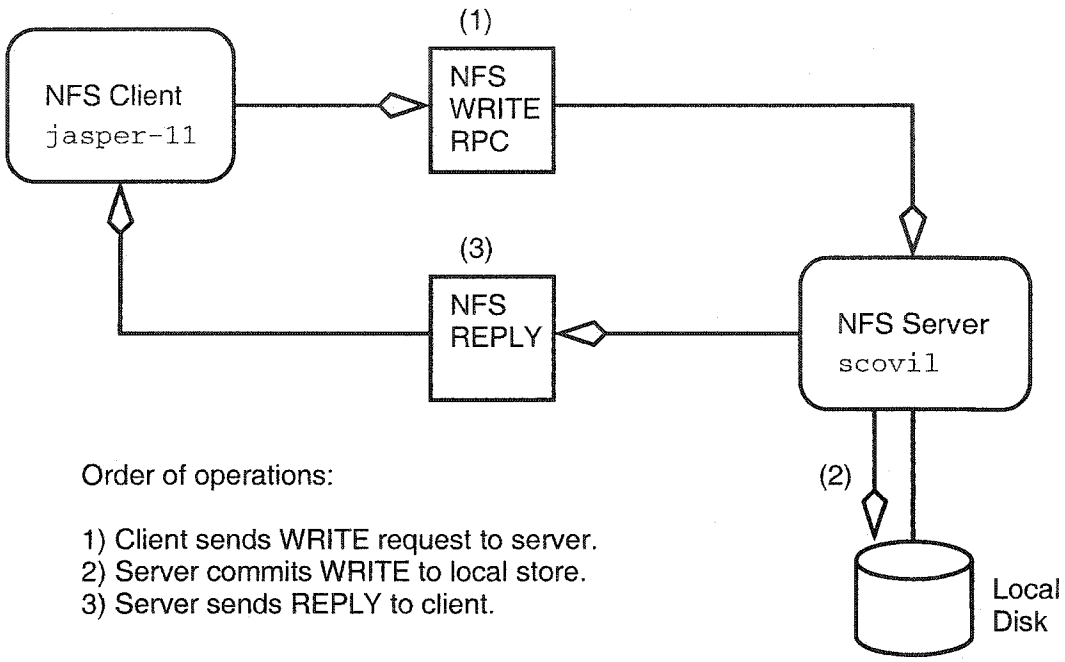


Figure 2.3: An illustration of the flow of operations involved in an NFS WRITE. The process on the client will block until a reply is received. See Figure 3.3 to compare an NFS write operation to a TrellisNFS write operation.

2.1.7 TrellisNFS Concepts

TrellisNFS is similar to NFS. With NFS, files on a central store are made available over a LAN. TrellisNFS is different in that rather than providing access to files already on the central store, files are copied on demand from a remote server into the local store. From there, the file is made available over NFS. The remote node that the file originated from is referred to as the file's *home node*. In other words, TrellisNFS usually involves three nodes.

2.2 Related Work

We now discuss some related work in the field of distributed file systems and distributed data storage systems.

2.2.1 The Coda File System

Coda is a distributed file system designed to work over a WAN. Coda allows for server replication, that is, storing the same files on multiple server to increase availability. Coda also allows for disconnected operation, in that a client can disconnect from the network and still have access to cached files. The files are synchronized with the server when the client is again connected to the network. Coda uses a cache management program called *Venus* that runs in user-space. Venus is responsible

for copying files in and out of the local Coda cache.

The Coda file system is implemented at the kernel level, and Coda implements all Unix file system functionality. It interfaces with a operating system's Virtual File System (VFS) layer, which allows seamless integration of a Coda namespace with the existing client file system hierarchy. Coda supports close-to-open consistency, and imposes a location-transparent namespace. Remote data storage servers cannot be dynamically added to the pool of available data storage servers; this design choice is integral to the security of a Coda system.

We considered using Coda as the mechanism to achieve seamless integration with the existing file system hierarchy. A Coda cache manager that uses the TrellisFS library to fetch and store files from a home node could be implemented. Using Coda clients instead of NFS clients would have some advantages; for example, a Coda cache manager knows when files are closed, so it would be easier to implement full close-to-open consistency across the entire metacomputer. We chose to not use Coda because most HPC consortia in Canada do not use it. Coda support is not generally enabled on HPC resources, and would therefore inhibit adoption of a TrellisCoda system.

2.2.2 The UFO file system

The UFO File system [4] is a distributed file system that runs completely in user-space. UFO uses the operating system's debugging mechanisms to catch system calls and redirect them to a user-space process. UFO supports transparent access to remote files stored on ftp servers and read-only access to files on HTTP servers, and it can be installed and used completely at the user level. However, UFO's system call interception mechanism is not generally portable across operating systems and imposes a non-trivial performance overhead.

UFO allows access to remote files through the FTP and HTTP protocols. Because of limitations in these protocols, full file system functionality cannot be supported. This limitation is not integral to UFO however, and other protocols can be used to implement full file system functionality. File system consistency in UFO is also dependent on the underlying protocol used for data transfer; the UFO file system caches remote files to the local disk. UFO implements a consistency policy based on a timeout. The timeout can be increased to provide weaker consistency, or shortened to provide stronger consistency. A timeout of zero is equivalent to close-to-open consistency. The security of a UFO file system is also tied to the underlying protocol. For example, with the FTP protocol, passwords can be stored in a special file so that they do not have to be entered as part of the file pathname. The UFO system supports 3 different methods of naming remote files: 1) through a URL, 2) through a filename that contains location and authentication information (because some applications are confused by URL syntax) and 3) through the notion of a UFO mount. To set up a UFO mount, the user associates a pathname prefix with specific protocol, location, and authentication information.

The UFO file system uses the debugging facilities of the operating system to implement a distributed file system. OS debugging facilities are specific to the operating system, and therefore UFO is not generally portable across operating systems. Like the TrellisNFS, the UFO file system uses a timeout to allow the user to control file system consistency strength. The overhead introduced by intercepting system calls is high for opening and closing files, but is relatively small for reading and writing.

2.2.3 The PUNCH Virtual File System

The PUNCH Virtual File System [13] (PVFS) is an NFS proxy that provides dynamic `uid` translation and can allow standard NFS clients to connect to standard NFS servers in a different administrative domain. PVFS provides on-demand, block transfer of file data; it does not provide additional caching over that which standard NFS clients already provide. PVFS does not require kernel changes, but does require super-user access in the administrative domains in which it runs. PVFS is designed to fit into the PUNCH environment.

PVFS implements full file system functionality. Since PVFS is only an NFS RPC forwarding mechanism, it supports close-to-open consistency. In terms of security semantics, PVFS inherits security from traditional NFS. It allows for dynamic translation of user identification credentials to permit interoperability among different administrative domains. Since PVFS uses the unmodified NFS clients and servers, its namespace is identical to that of a traditional NFS system. The establishment and disconnection of a PVFS hierarchy, and therefore location information, is not controlled by PVFS itself but through PUNCH middle-ware. PVFS achieves seamless integration through its use of NFS.

PVFS only rewrites NFS packet contents, it does not batch them. Because NFS remote procedure calls are synchronous, using PVFS over a high latency network, such as the Internet, will result in unacceptable performance. TrellisNFS uses whole-file caching to reduce the number of remote procedure calls. PVFS is useful only when combined with PUNCH middle-ware. Since the PUNCH middle-ware is responsible for establishing and disconnecting NFS mounts, it must run as a root process.

2.2.4 The Legion NFS Server

The goal of the Legion project [14] is to incorporate sparsely connected computing resources under a single virtual supercomputer with a single system image. It addresses the notion of a global file system through its Legion I/O libraries and NFS server [29]. The Legion I/O model focuses on performance and usability, the Legion I/O library provides a hierarchy of classes for interacting with the Legion virtual supercomputer. The Legion project has implemented an NFS server that allows a Legion file system hierarchy to be integrated with the file system hierarchy of the local machine. The

NFS server uses an asynchronous read-ahead and write-behind cache for remote data. The Legion I/O libraries are most efficient when operating on data in large granules.

The Legion I/O libraries provide full file system functionality. When using the Legion NFS server, written data is flushed from the server's cache after a configurable delay. In addition to supporting the same security mechanisms as traditional NFS, the Legion NFS server and the NFS client are co-located on the same machine. The Legion NFS server accepts connections only from the local loop-back network interface, this prevents malicious users from snooping NFS traffic on the LAN, as well as preventing an attack based on spoofing an IP address. The Legion I/O model imposes a namespace known as *context space*. This namespace does not require location information in the name for a file to be located.

In some ways, the architecture of the Legion system is similar to that of the Trellis system. Both implement a library for file access in a virtual supercomputer, and both implement an NFS server to integrate their namespace into the local file system hierarchy. Unlike the Legion I/O Library, the TrellisFS Library supports two modes of file access, which mode to use depends on the nature of the application's workload. The Legion I/O model uses only read-ahead and write-behind caching. To enhance security, the Legion NFS server will only accept client connections from the same host (i.e., 127.0.0.1), and therefore employs a one-server-per-client ratio. This way, forged NFS packets, with spoofed headers, will not be accepted by the server. TrellisNFS clients are typically compute nodes in a cluster; the system already has security policies and contracts with the users. Root access to restricted, and non-root users cannot forge IP packets; also, the cluster is on a private subnet so a forged IP packet from the Internet is not a risk. The Legion NFS server must run as a root process, by keeping the TrellisNFS server running as an unprivileged process we provide additional security from programming bugs or any unforeseen consequences of integrating the file system hierarchy of a foreign administrative domain. The current implementation of the TrellisNFS system does not support a location-independent namespace, we chose this because we do not foresee a location-independent namespace as an advantage for the Trellis environment; this is because in a typical HPC environment users know on what server they have stored their data.

2.2.5 Secure NFS

Secure NFS [7] (SNFS) uses the Secure Shells port forwarding mechanism to forward NFS messages through a secure tunnel. The mechanism to accomplish this is the User Datagram Protocol (UDP) forwarding mechanism of SSH (version 2) and an additional software program that must be installed on top of SSH. To set up SNFS, an NFS server exports mount points to itself. The NFS client will mount an NFS volume from itself. SSH and the SNFS RPC forwarding program are the connective services that forward client and server requests to each other.

Since SNFS is based on NFS, it implements full file system functionality. Also, like NFS,

SNFS supports close-to-open consistency. Security in SNFS is inherited from NFS, with the added protection of encrypting packets that pass over the network. Since SNFS uses the unmodified NFS client, it seamlessly integrates with client's local file system hierarchy.

NFS requires root privileges to install and root privileges on both the client and server machines to set up. However, once it has been set up, any user can access files on the NFS server without additional system administrator support. SNFS shows a slight performance decline over unsecured NFS when operating on a LAN. There has been no evaluation of SNFS' performance when operating over the Internet, although since NFS (and therefore SNFS) uses synchronous remote procedure calls, using SNFS over the Internet would result in unacceptable performance.

2.2.6 The Trellis File System

The Trellis File System [25] is a library that allows access to remote files accessible through SSH, HTTP, or FTP. The TrellisFS namespace supports either Secure Copy Locater (SCL) or Uniform Resource Locater (URL) syntax. TrellisFS is an overlay file system, meaning that it: 1) does not require special kernel support, 2) does not require super-user permission to install and 3) does not require users to share the same namespace, although users can share a common namespace if they wish. TrellisFS is built on top of existing file system services, and provides a C language interface that mimics the POSIX file system API. TrellisFS supports whole-file caching, sparse-file access and file data pre-fetch.

The original version of TrellisFS only supports functions to access file data; file metadata, directories, hard and symbolic links, and file renaming were not supported. We have added this functionality as part of this work. TrellisFS supports close-to-open consistency. All access to remote data is done through the SSH, which provides end-to-end encryption. Also, TrellisFS uses the authentication and authorization mechanisms provided by the SSH. As mentioned above, TrellisFS supports SCL and URL syntax in its namespace. It also supports a location-independent method of naming files through the use of Unix environment variables. However, the TrellisFS library does not seamlessly integrate with an existing file system hierarchy. Applications that wish to integrate with TrellisFS must modify their source code and be re-compiled.

The TrellisFS library is an integral part of the TrellisNFS system. It could be said that the TrellisNFS user-level server is the glue that allows seamless integration of a TrellisFS namespace with an existing client file system hierarchy. As part of this work, we made many improvements to the TrellisFS library and its related components.

2.2.7 The Ivy File System

The Ivy file system [19] is a peer-to-peer file system. Ivy provides three novel contributions over previous work in peer-to-peer file systems: 1) Ivy supports multiple readers and writers; 2) it does

not require that all users of the file system fully trust each other; and 3) it uses a distributed hash table to support replication and high availability. The Ivy file system is a log-based file system, similar to transaction logs in a database system. When a user performs a file system operation, Ivy scans a chain of log records to satisfy the request. File system integrity is maintained through the use of public key encryption.

The Ivy file system uses an NFS server to integrate with the client's existing file system hierarchy. The Ivy NFS server modifies the NFS client by adding a close remote procedure call, this was done in order to support close-to-open consistency. Since the Ivy file system replicates log entries on multiple potentially untrusted computers, security can only be enforced by requiring the user to encrypt their files. Because of this method of distributing log entries across potentially untrusted computers, Ivy does not enforce ownership or permission modes of files.

The Ivy file system is designed to work in a peer-to-peer setting where the respective users may not fully trust each other. These design goals differ from the ones we list in Chapter 1. In addition, we feel that modifying the NFS client is not an acceptable option for the Trellis environment. The NFS client is implemented in the operating system kernel; requiring that system administrators of participating sites modify their operating system's kernel is unreasonable and will inhibit adoption of the file system.

2.3 Concluding Remarks

In this chapter we discussed the differences between a distributed data storage system and a distributed file system. A distributed file system provides more functionality than a distributed data storage system.

We also reviewed some previous work in the field of distributed file systems; however, of the works reviewed, none matches the goals we have laid out in Chapter 1. Fully functional distributed file systems such as Coda and the Legion NFS server require more system administrator assistance to install and use than should be necessary. Distributed file systems such as PVFS and SNFS will not perform well over a high latency network like the Internet. The Ivy file system is designed to work in a peer-to-peer environment, where the different peers may not trust each other. The goals set out by the Ivy designers make Ivy unsuitable for use in the Trellis environment. The UFO file system is generally not portable and the system call redirection mechanism is expensive.

Chapter 3

The Trellis Network File System Architecture

In the previous chapter, we discussed some important background concepts that relate to building a distributed file system. We also commented on previous work in the field that relates to the TrellisNFS server. Now, we discuss the architecture of the TrellisNFS server. The complete TrellisNFS system with all related components is shown in Figure 3.1. The three principal components of the TrellisNFS server are illustrated in Figure 3.2. They are: 1) the user-level server, 2) the Trellis file system library, and 3) the SSH Proxy. The focus of this chapter is on these three components.

3.1 The NFS Client

In the TrellisNFS system, the NFS client is unmodified. This design decision is important because, in theory, it allows our system to be compatible with the existing deployment of NFS clients. We did not implement our own NFS client, as one of the goals of the project is to be compatible with, and take advantage of, existing infrastructure as much as possible. The current wide use of NFS is the principal reason we chose it as the protocol of choice to achieve seamless integration of the Trellis File System with an existing file system hierarchy. Other members of the Trellis team are investigating the integration of the TrellisFS library with the Samba server [3], as an alternative to NFS.

NFS clients are implemented in the operating system's kernel. If we chose to modify the NFS client, as does the Ivy [19] file system, we would need to ask system administrators to modify their operating system kernel to use the TrellisNFS system. In many cases this is not possible, proprietary operating system vendors are generally not willing to allow users access to their operating system's source code. In all cases it is inconvenient for system administrators to make kernel changes.

System administrator involvement in setting up an HPC cluster to use the TrellisNFS server is minimal. For example, as part of the CISS-3 experiment, we configured TrellisNFS for use on the

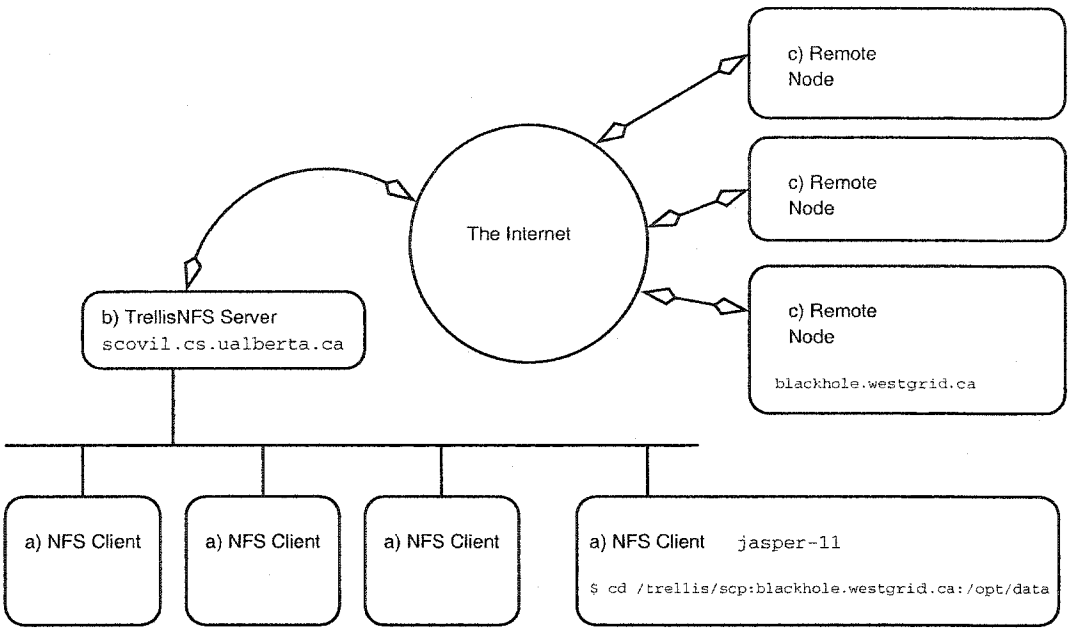


Figure 3.1: The complete TrellisNFS system with all related components. These components are a) the NFS client, b) the NFS server and c) remote data storage server.

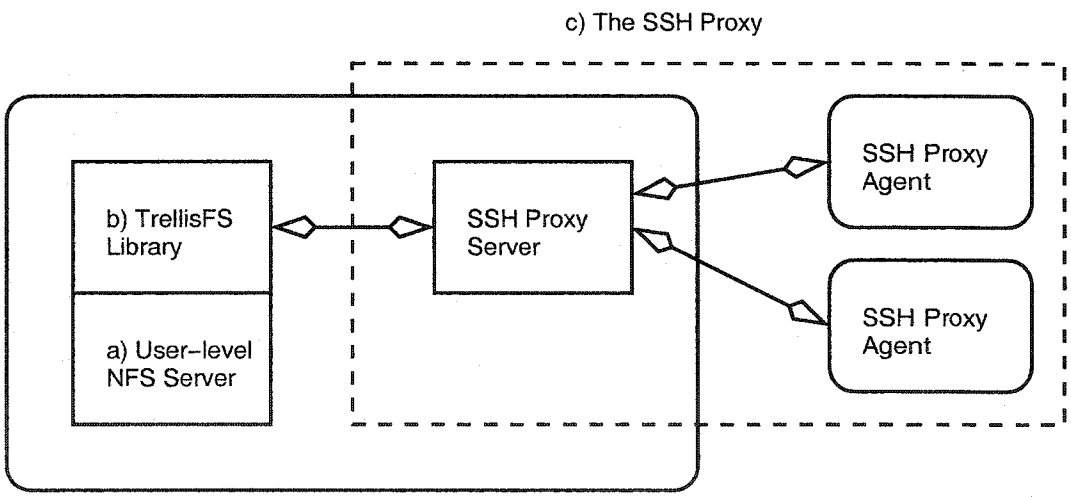


Figure 3.2: The main components of the TrellisNFS server. These components are a) the user-level server, b) the TrellisFS library and c) the Secure Shell Proxy.

chorus cluster at the University of New Brunswick, the only system administrator support required to install the TrellisNFS server was appending a single line to the file system table (i.e., the mount table) of all the nodes in the cluster. We were able to compile, install, and run the server; create server and client mount points; and establish the mount from our unprivileged user account.

Protocol compliance does not always guarantee compatibility with existing client implementations. We tested the NFS client in the Irix operating system with the TrellisNFS server, and found that file system operations other then creating, reading and writing files were not possible due to implementation differences.

3.2 The TrellisNFS Server

The TrellisNFS user level server is the front end of the three components we discuss in this chapter. The TrellisNFS server is based on Linux's UNFSD server [24]. We have modified the original server to allow it to work with remote files. The primary change was integrating the application with the TrellisFS library (we discuss the TrellisFS library in Section 3.3). In addition to this integration, the semantics of working with remote files required additional changes to the NFS server. In this section we discuss how the architecture of the original server was modified. Recall that an NFS server system consists of two servers, the MOUNT server and the NFS server.

There are three key differences between NFS and TrellisNFS:

1. *Dealing with network latency*: With NFS, the client and server are connected via a LAN, and the server's files are on its local disk (refer to Figure 2.2). A LAN has higher bandwidth and lower latency then a WAN. The timeout/retry algorithms in NFS clients are tuned to LAN latencies. The TrellisNFS server works with files on the Internet (refer to Figure 3.1). On a high latency network, NFS performance can degrade significantly due to client timeouts and retries. TrellisNFS uses aggressive caching to offset high WAN latencies.

2. *Device, I-node and IP Address name conflicts*: The files available from a single NFS volume come from the same device. Servers that provide data from multiple devices (for example, from multiple partitions or disks) do so by exporting one NFS volume per device. The TrellisNFS server makes files from multiple servers and multiple devices available through the same NFS volume. NFS servers have this policy because an NFS server must provide a unique NFS file-id number for each file it exports; Unix NFS servers use the i-node number assigned by the underlying file system as the file's file-id number. If a server were to export multiple devices, there would be the possibility of two files having the same identifier. The TrellisNFS server exports multiple files from multiple servers, each with multiple devices, and must avoid file-id collisions. See Section 4.1.3 for a discussion on how TrellisNFS generates unique file-id numbers.

3. *Dynamic integration of home node file system hierarchies*: In a traditional NFS setup, the system administrator declares available NFS file systems statically via system configuration files. With TrellisNFS, the system administrator must add the Trellis mount point to the client's file system table, but the connection to a remote node is made dynamically, on-demand, when a user tries to access a file on a remote node [13].

The TrellisNFS MOUNT server is unmodified from the original. Recall that the purpose of the MOUNT server is to give the root file handle to the client, and to maintain a list of which clients have mounted the NFS volume.

3.2.1 The NFS server

The TrellisNFS server runs in single-threaded mode under an unprivileged user account. Modifications to the NFS server can be grouped into five categories. We will discuss these modifications now:

1. *File System API*: The original user level server called Unix file system API functions directly. We modified call sites in the original server to call file system functions from the TrellisFS library.

2. *Namespace*: The original server only recognized files from the Unix file system namespace. We modified the server to recognize Trellis SCLs.

3. *NFS File Handle Format*: The format of the NFS file handle has changed to store additional information needed to support remote file system access. Also, the method of rebuilding an SCL from the information contained in the NFS file handle has changed. Recall that NFS clients do not interpret the contents of an NFS file handle. NFS file handles are opaque to the NFS client.

4. *Consistency*: We define file *consistency* to be the notion of where the latest version of a file is considered to be located. After a NFS WRITE procedure call, the file in the Trellis Cache is the most recent version. During this time, if a third party reads the original file on the home node, they will not see the latest version. This window of inconsistency between the cached copy of a file and the copy on the home node implies weak file consistency; this weak consistency is intentional. With the original server, file changes were committed to disk frequently; however, with the TrellisNFS server, frequent synchronization of file data between cached and remote files would waste bandwidth.

   The NFS protocol requires that data from a WRITE procedure call be committed to stable storage before the procedure call returns. The original user-level server did not comply with this requirement. After a write, data is held in the kernel buffer cache, and the operating

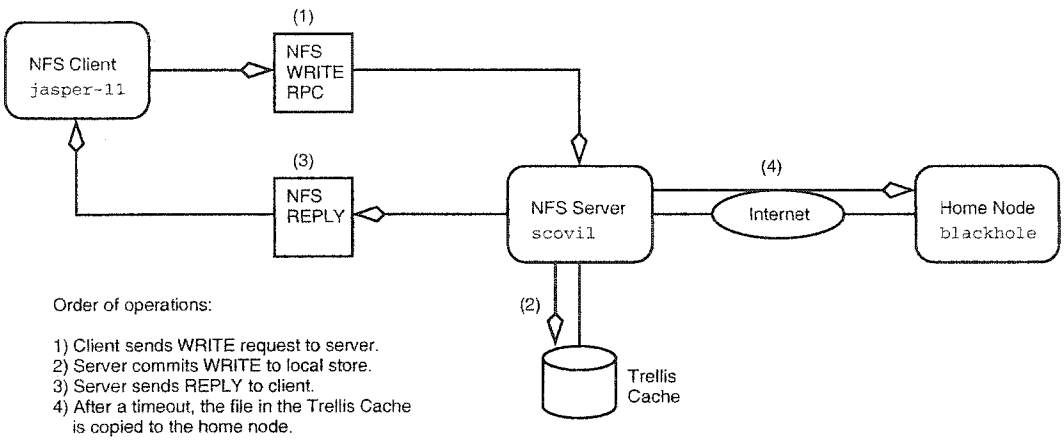


Figure 3.3: An illustration of the flow of operations involved in a TrellisNFS WRITE. The process on the client will block until a reply is received. See Figure 2.3 to compare a TrellisNFS write operation to an NFS write operation.

system schedules the actual disk write at a future time. TrellisNFS maintains this semantic; it also does not commit the write to the file on the home node before returning from the WRITE procedure call.

Because the NFS protocol is stateless, it does not contain a procedure call that indicates to the server when a client application has closed a file. Having information about when an application closes a file would help the NFS server to determine when to synchronize a file in the Trellis cache with the original copy on the home node. The TrellisNFS server will synchronize dirty data in the Trellis Cache after a timeout. Refer to Figure 3.3 for an illustration of a TrellisNFS write operation.

5. *Unique File-Id Generation*: The original server was designed to be able to export files from multiple devices on the same node. The TrellisNFS server exports files from multiple remote nodes, each with potentially multiple devices. Because the server exports from multiple nodes in addition to multiple devices, the method of generating file-id numbers was modified.

3.2.2 Crash Recovery

The stateless model of the NFS protocol makes recovering from a server crash simple. It is possible for the server to crash and restart without the client knowing. Every NFS request contains an NFS file handle that uniquely identifies a file on the server. The NFS server examines the NFS file handle to determine which file the client is requesting. The contents and format of a file handle cannot change across server restarts.

NFS operations are synchronous. For example, an NFS write must be committed to stable storage before the server can return a result to the client. This way, if the server crashes in the middle

of processing an NFS request, no result will be returned to the client, and the client will retry the request. Once the server restarts, the clients request can be serviced.

Another reason crash recovery is simple is because the NFS protocol makes most procedure calls idempotent. Idempotent requests can be executed multiple times and the result is always the same. Some NFS procedure calls, such as REMOVE, are non-idempotent by nature. It is possible for unexpected operation to occur when processing a non-idempotent request. For example, a client requests that a file be removed, and the server crashes after the file is removed, but before the reply is sent. In this case, the server will restart, and the client request will be re-tried. Since the file no longer exists, the server will incorrectly return an error [10].

3.3 The Trellis File System Library

The Trellis File System library implements a Unix file system API [25]. The TrellisFS library is semantically equivalent to the Unix API, except that the TrellisFS library allows access to remote files, not just the files in the local file system hierarchy.

The TrellisFS library is implemented using the concept of a function *wrapper*. For example, the function `trellis_open()` will fetch a remote file into the Trellis cache, and call the local Unix `open()` function.

The original implementation of the TrellisFS library provided functions to access and manipulate file data. We extended the library to handle file metadata, directories, directory metadata, links and file renaming. The complete list of functions supported by the TrellisFS library, with their Unix equivalents, is provided in Table 3.1. It is our goal to make these functions as semantically equivalent to their Unix counterparts as possible. TrellisFS allows access to files on remote file systems that can be reached via the Secure Shell. Although TrellisFS provides access to remote files through a variety of protocols, such as SSH, HTTP and FTP, for the purposes of the TrellisNFS server we exclusively use SSH.

The file system functions supported by the TrellisFS library can be divided into 5 categories. 1) File functions, 2) directory functions, 3) metadata functions, 4) hard and symbolic link functions and 5) user authority functions. In addition to file system functions, additional functions have been implemented to facilitate working with remote files.

TrellisFS uses *whole-file* caching. Cached files are placed on the local disk in a special directory called the *Trellis cache*. File data and some metadata are stored in the Trellis cache.

TrellisFS also supports *sparse file access*. In sparse file access mode, only the portion of the file the user is interested in is fetched from the home node, and not the entire file. The TrellisNFS server does not use sparse access mode.

3.3.1 API Details

We now discuss how the semantics of TrellisFS functions differ from the semantics of standard Unix file system functions. We group our discussion according to the 5 function categories listed above.

1. *File Functions*: The functions to open, close, read and write files are the core TrellisFS functions. They are semantically equivalent to their Unix counterparts. On a call to `trellis_open()` the remote file is fetched and opened. If a copy of a file already exists in the Trellis cache, its MD5 [22] checksum is compared with the MD5 checksum of the original file; if the checksums are the same, the file is not fetched. On a call to `trellis_close()` the file is closed and, if necessary, resynchronized with its remote copy. Calling `trellis_close()` implies potentially copying data over the network.

   If we know there will be additional changes to the file and wish to delay file synchronization until later, the API provides the functions `trellis_close_no_flush()` and `trellis_reopen()`. The former will close the file, but not copy a changed file back to its home node. This is useful, for example, if we know the file will be deleted immediately after it is closed. The later allows the user to open the same file with a different mode, for example, for writing instead of reading, without synchronizing the file with its remote copy.

2. *Directory Functions*: Since the TrellisFS library does not cache directories, directory functions involve remote communication.

   The semantics of the `trellis_readdir()` function differ from those of the Unix `readdir()` function. The Unix `readdir()` function usually returns the same i-node number as the Unix `stat()` function. The i-node returned with the `trellis_readdir()` function is the i-node of the remote file, as returned by the remote `readdir()` function. This is different from `trellis_stat()`, which returns the i-node of the file in the Trellis cache. We discuss our choice to return the i-node of the cached file rather then the remote file in Section 4.1.3.

3. *Meta Data Functions*: The metadata functions query metadata of cached files, and remote directories. Metadata fields supported in a Unix file system are shown in figure 3.4. There are also functions to query the metadata of a remote file if the user requires it. We choose to query cache file metadata where possible for performance reasons (see Section 4.1.3 for further discussion). Since we query remote data for directories and local data for files, there is the possibility that a file and a directory could have the same device and i-node number. This differs from Unix file system semantics.

   Metadata functions return ownership data that is valid only on the file's home node. The TrellisFS library provides a means to automatically map user and group ownership informa-

```
/* From section 2 of the Linux manual ("man 2 stat"). */

struct stat {
    dev_t     st_dev;         /* Device number */
    ino_t     st_ino;         /* I-node number */
    mode_t    st_node;        /* Permission bits */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device type (if i-node device) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for file system I/O */
    blkcnt_t  st_blocks;      /* Number of allocated blocks */
    time_t    st_atime;       /* Time of last access */
    time_t    st_mtime;       /* Time of last modification */
    time_t    st_ctime;       /* Time of last status change */
};
```

Figure 3.4: A C-style structure showing the metadata fields available on a Unix file system [1].

tion from the file's home node to equivalent information on the local node. This *uid and gid mapping* is discussed in detail in Section 4.2.2.

4. *Hard and Symbolic Link Functions*: Hard and symbolic link functions query remote files; calling these functions will result in a remote operation. All other semantics are the same as those of the traditional Unix file system API.

5. *User Authority Functions*: The user authority functions are not file system functions, but are used by the TrellisNFS server to enforce security. These functions are identical to their Unix counterparts except that they operate on a remote node. Calling these functions will result in a remote operation.

The TrellisNFS server and the Trellis File System are designed for HPC workloads. Many file system functions such as directories, symbolic links, and the persistent semantics of i-nodes are not used by HPC applications.

3.3.2 The Meta Data Cache

The TrellisFS library has an optional metadata cache. The metadata cache stores the results of `trellis_stat()` or `trellis_lstat()` calls. A consequence of the metadata cache is that external updates to a remote file system may not be seen through TrellisFS. The benefit of the metadata cache is that it significantly reduces remote communication from calls to `trellis_stat()` or `trellis_lstat()`. The metadata cache provides a significant performance boost to applications, such as the TrellisNFS server, that frequently query metadata. See Section 5.3 for experimental results.

TrellisFS Function	Unix Function	Locality	Comments
File access functions			
trellis_open()	open()	remote	
trellis_fopen()	fopen()	remote	
trellis_reopen()		local	
trellis_close()	close()	remote	
trellis_close_no_flush()		local	Local and remote files are not synchronized.
trellis_fclose()	fclose()	remote	
trellis_read()	read()	local[1]	
trellis_write()	write()	local[1]	
trellis_lseek()	lseek()	local[1]	
trellis_truncate()	truncate()	local	
trellis_flush_all()	fflush(), fsync()	remote	All remote files are synchronized with cached files.
trellis_close_all()	close()	remote	Same as flush_all, any open files are no longer valid.
User authority functions			
trellis_getuid()	getuid()	remote	
trellis_getgroups()	getgroups()	remote	
Metadata functions			
trellis_stat()	stat()	local or remote[2]	
trellis_stat_remote_only()		remote	
trellis_lstat()	lstat()	local or remote[2]	
trellis_lstat_remote_only()		remote	
trellis_unlink()	unlink()	remote	
trellis_unlink_remote_only()		remote	
trellis_utimes()	utimes()	remote	
trellis_chmod()	chmod()	remote	
trellis_chmod_remote_only()		remote	
trellis_lchown()	lchown()	remote	
trellis_rename()	rename()	remote	
Hard and Symbolic Link functions			
trellis_link()	link()	remote	
trellis_symlink()	symlink()	remote	
trellis_readlink()	readlink()	remote	
Directory Functions			
trellis_mkdir()	mkdir()	remote	
trellis_rmdir()	rmdir()	remote	
trellis_opendir()	opendir()	remote	
trellis_readdir()	readdir()	remote	
trellis_closedir()	closedir()	remote	
trellis_seekdir()	seekdir()	remote	
trellis_telldir()	telldir()	remote	

1 — The Operation could involve remote communication if sparse file access is enabled.
2 — If the file is open the operation is local.

Table 3.1: The Trellis File System API with related Unix API functions

The Metadata cache is a general purpose solution to the problem of redundant metadata queries. Calls to the Unix file system API functions `stat()` and `lstat()` are fast on a local file system; therefore, existing applications will issue multiple calls to these functions rather then store and re-use the results of the first call.

3.4 The Secure Shell Proxy

The Secure Shell proxy is distributed as part of the TrellisFS library. The original implementation of the SSH Proxy is due to Siegel and Lu [25].

Figure 3.5 shows the architecture of the SSH Proxy. TrellisFS communicates with remote nodes to perform file system operations. Setting up a new SSH connection costs some overhead. To avoid repeatedly paying this overhead, it is desirable to maintain a *persistent* connection. This is the primary function of the Secure Shell Proxy.

Another function of the SSH Proxy is to implement a convenient way to execute remote procedure calls over SSH on a remote node. For example, a user program calls `trellis_mkdir()` on a local node; the TrellisFS library sends a message using the SSH Proxy to a remote node instructing it to execute the its `mkdir()` function and return a result code, which the library then returns to the user program.

The SSH Proxy consists of 3 main programs: the client, the SSH Proxy server, and the SSH Proxy agent. The SSH Proxy server maintains persistent connections to remote nodes and relays messages between clients and remote nodes. The SSH Proxy agent is a program that runs on the remote node. It accepts, processes and replies to messages sent via the SSH Proxy server. The client initiates SSH Proxy requests by communicating with the SSH Proxy server running on the local node.

The SSH Proxy suite comes with two standard clients, `ssh_via_proxy` and `scp_via_proxy`. These programs mimic the functionality of the `ssh` and `scp` programs, respectively.

3.5 The Trellis Security Infrastructure

The Trellis Security Infrastructure [17] (TSI), allows single sign-on (SSO) capability using only SSH and SSH agents. SSO means that a user needs to authenticate to the overlay metacomputer only once. The TrellisFS library uses the TSI for authentication and authorization with remote nodes. The user can then access resources from all other nodes in the metacomputer without re-authenticating. This is a benefit to the TrellisNFS server; the server can set up TSI authentication once, and then access file systems from anywhere in the metacomputer without the need to enter passwords or pass phrases.

The TSI is similar to the Grid Security Infrastructure [9] (GSI) in its basic design goals. The

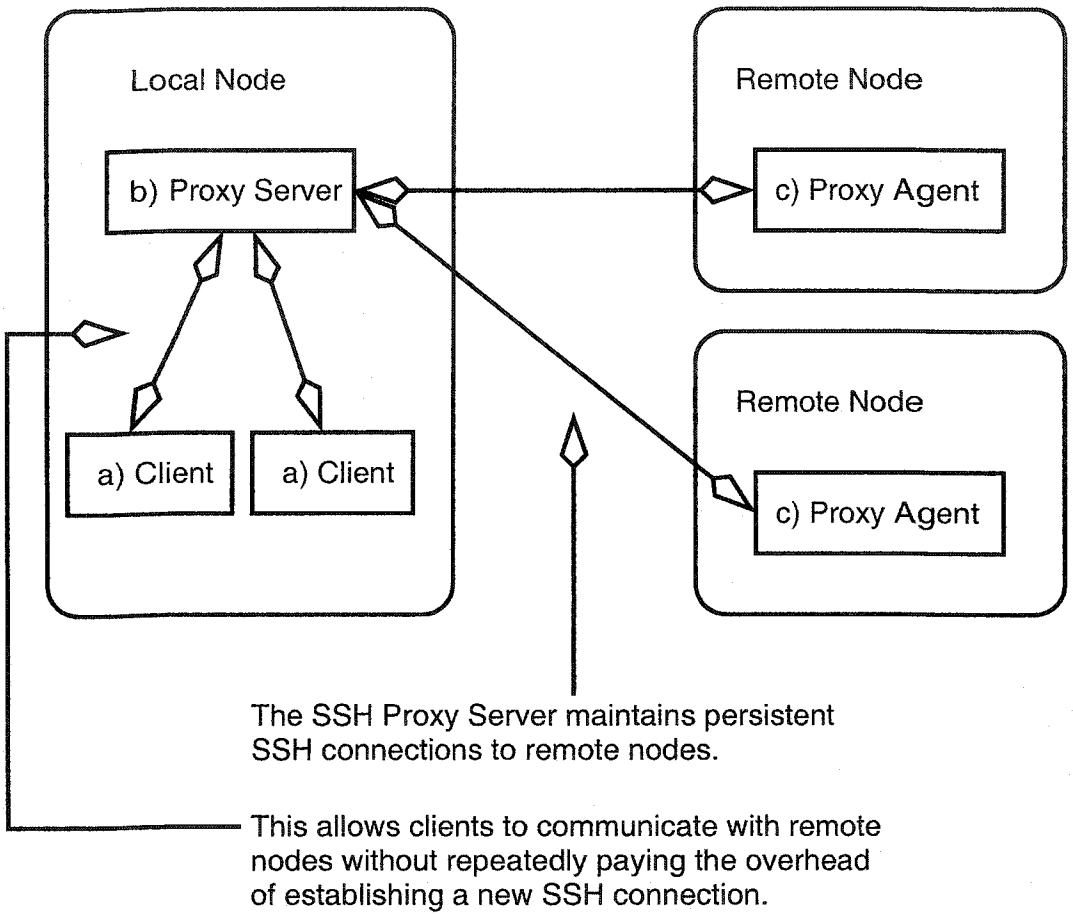


Figure 3.5: An architectural diagram of the SSH Proxy. The main components are a) the client, b) the server and c) the agent. The core function of the SSH Proxy is to maintain persistent SSH connections to remote nodes, allowing clients to send and receive messages to and from remote nodes without the repeated overhead of setting up a new SSH connection.

difference is that the TSI shifts the burden of administration from the system administrators to the users. This has the advantage that system administrator involvement is not required to install and configure the TSI; however the user must now manage administration tasks themselves.

The TSI includes a tool called *trellis-ssh*. Trellis-ssh is a drop-in replacement for the standard `ssh` command; it allows for SSO operation and allows the user to access remote nodes through a gateway. Trellis-ssh is a wrapper around the standard `ssh` command.

3.6 Security

In this section, we talk about the security issues relating to the TrellisNFS server. We first discuss security between NFS clients and the TrellisNFS server. Next, we discuss security between the TrellisNFS server and the home nodes.

3.6.1 NFS Security

Security issues between NFS clients and the TrellisNFS server are identical to the issues found in a traditional NFS setup. All NFS traffic is unencrypted over the local network. A user inside the local administrative domain could potentially monitor and record these unencrypted NFS packets. Computing clusters are typically implemented on a private subnet, where network packet monitoring is only possible by a system administrator.

The TrellisNFS server controls access to its volumes on a *per-host* granularity. The server can determine if a request originated from a *secure port*. A secure port has a port number less than or equal to 1024. By allowing connections only from secure ports, the server prevents ordinary users from communicating with the server by generating custom NFS requests that pretend to originate from a different user, and therefore compromise security.

The NFS client also plays a role in NFS security. The NFS client enforces security at the *per-user* granularity. For example, the NFS client will prevent user Bob from accessing files belonging to user Alice. A consequence of this NFS design choice is that when the server gives access to a client to mount its NFS volume, any user with super-user access to that client can access *all* non-root owned files exported by the TrellisNFS server.

The current implementation of the TrellisNFS server supports only a single user. The user who wants to access files over TrellisNFS must run the server under their account. Any files on a remote node that the user does not have access to will be reported as being owned by the special user *nobody*.

One consequence of our policy to query the metadata information of files in the Trellis cache is that all files (but not directories) will appear to be owned by the user running the server. However, if the user does not have access to the file on the remote node, any attempt to read or write to the file will fail.

In a traditional NFS setup, a file with world read permissions can be read by users in the local administrative domain. With the TrellisNFS server, the model changes somewhat. As an example, refer to Figure 3.1: a TrellisNFS server on machine `scovil` at the University of Alberta is used by the NFS client `jasper-11` to access files from machine `blackhole` at Simon Fraser University. If a user on `blackhole` makes a file world readable, then all users on `jasper-11` and other NFS clients of the TrellisNFS server running on `scovil`, in addition to all users on `blackhole`, can now read that file. To help preserve the original security model, the TrellisFS library can optionally zero group or world permissions on all files and directories.

In a traditional NFS setup, the system administrator has control over both the client and the server. With TrellisNFS, the remote node, and files on the remote node are not under the control of the local administrator. A malicious user could exploit this by installing a set-uid binary on a remote node. NFS server options such as *root squashing* [26] and configuring a client to disallow *set-uid execution* should be enabled to prevent this type of attack.

3.6.2 Security of over-the-Internet traffic

The TrellisNFS server uses the Trellis Security Infrastructure [17] for access control, authentication and authorization. All over-the-Internet traffic is encrypted with the SSH. The same access and authentication mechanisms built into the SSH are available for controlling access and authentication between the TrellisNFS server and a remote node.

The TrellisNFS server uses the SSH public-key authentication method. Private keys are stored in a SSH agent, and the server uses this agent to gain access to remote systems. Granting and revoking access to remote systems is a matter of managing the private keys loaded in the server's agent, and placing public keys in the `authorized_keys` file on the home node [5].

3.7 Concluding Remarks

In this chapter, we discussed the architecture of the TrellisNFS system. We designed the TrellisNFS server to be used in the Trellis environment, a large scale metacomputing environment. The TrellisNFS server provides applications with seamless, transparent access to files on a remote data storage site.

The TrellisNFS system consists of four components: 1) the NFS client, 2) the TrellisNFS server, 3) the TrellisFS library, and 4) the Secure Shell Proxy.

We chose not to modify the NFS protocol, and we have not intentionally changed semantics that NFS clients expect from NFS servers. We discussed architectural changes that were made to Linux's UNFSD server to deal with working across multiple administrative domains, and also with the high latencies of the Internet. We discussed how to preserve the NFS model of crash recovery.

We discussed the necessary extensions made to the TrellisFS library in order that it would be able to support the full Unix file system API. We also discussed the SSH Proxy, a high-performance architecture that allows remote command invocation, remote data copying and execution of remote procedure calls.

We finished the chapter by discussing the security of the TrellisNFS system. We identify known security issues inherent in NFS. Since we decided not to implement our own NFS client or modify the NFS and MOUNT protocols, we inherit all the same security considerations associated with a traditional NFS server.

Chapter 4

Implementation Details

In the previous chapter, we discussed the architecture of the TrellisNFS server. In this chapter, we discuss the implementation the TrellisNFS server and the TrellisFS library. We exclusively used the Linux NFS client during the development of the TrellisNFS server.

4.1 The TrellisNFS Server

Figure 4.1 shows a complete architectural diagram of a TrellisNFS configuration. For the remainder of this section we will discuss the MOUNT server, modifications we made to the original user-level NFS server, and issues in preserving crash recovery.

4.1.1 The MOUNT server

There are four details relating to the MOUNT server that we will discuss: 1) The MOUNT server is able to operate using an unprivileged port; 2) It contributes to the security of TrellisNFS; 3) It maintains a list of clients that have mounted an NFS volume; and 4) The MOUNT server generates the root NFS file handle and provides it to NFS clients.

The MOUNT protocol uses a port-mapper assigned port, the port number can be in the range of port to which an unprivileged process can bind. If this were not the case, we could not run our server as an unprivileged process.

The MOUNT server plays a role in NFS security. An NFS server system administrator lists the host names of nodes that have access to the NFS volume in the servers *exports* file. The exports file is typically located at `/etc/exports`, and lists whether a client has read-only or read-write access. When a MOUNT request comes in, the MOUNT server checks the client's IP address against the contents of the exports file. A typical exports file looks like this:

```
/usr/scratch/trellis    jasper-11(rw),jasper-12(rw)
```

In this example, the server has given the clients `jasper-11` and `jasper-12` permission to mount

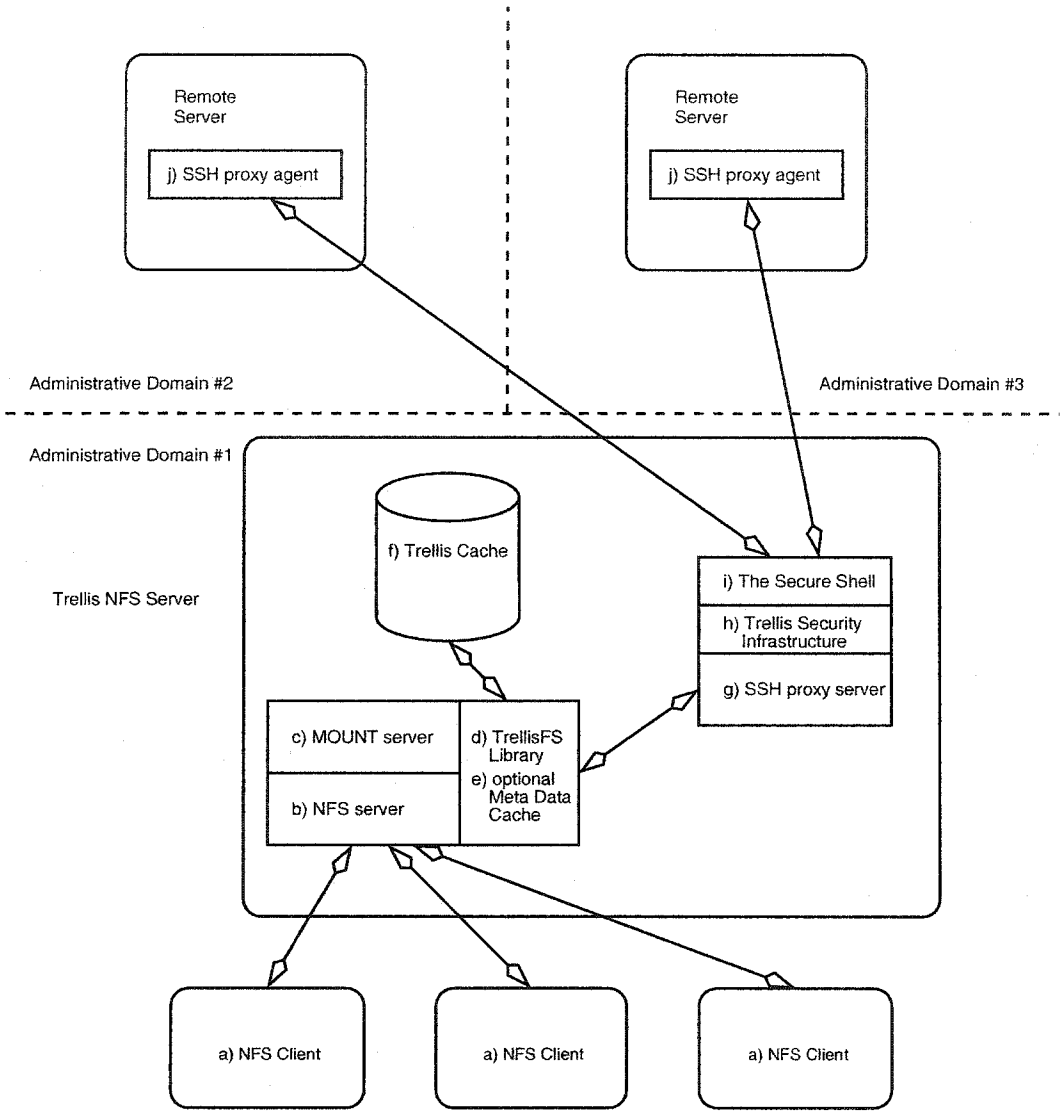


Figure 4.1: A complete architectural diagram of the TrellisNFS system. Components include: a) the NFS client, b) the NFS user-level server, c) the MOUNT server, d) the TrellisFS library, e) the optional meta data cache, f) the Trellis cache, g) the SSH Proxy server, h) the Trellis Security Infrastructure, i) the Secure Shell and j) the SSH Proxy agent.

the NFS volume at `/usr/scratch/trellis`. The `(rw)` means the client has read-write access to the NFS volume.

As an additional security check, the MOUNT and NFS servers check the incoming port number of the client. If the port number is in the range of ports to which a standard user can connect, the connection is denied. This mechanism prevents a user from accessing another user's files by constructing a custom RPC packet with forged authentication information. There are known security issues relating to NFS that can be managed, for more information the reader is referred to Stern [26].

The MOUNT server maintains a list of clients and the NFS volumes each client has exported. Tracking this state information is the reason that the functionality of the MOUNT protocol was not integrated into the NFS protocol [10]. The MOUNT server writes this state information to a special file called `rmtab`. The `rmtab` file is usually located at `/etc/rmtab`. A typical `rmtab` file looks like this:

```
jasper-11:/usr/scratch/trellis
jasper-12:/usr/scratch/trellis
```

Clients `jasper-11` and `jasper-12` have each mounted the TrellisNFS volume. The directory `/usr/scratch/trellis` is used as the root mount directory.

The MOUNT server also passes the root NFS file handle to the client. The code to generate this file handle was not modified from the original server's code. A directory must exist on the server before the root file handle can be generated; this directory is used when generating the root NFS file handle. When a STATFS procedure call is executed, information from the file system that this directory resides in, is used in the reply. File handle generation is discussed in detail in Section 4.

4.1.2 The original user-space NFS server

The TrellisNFS server is based on Linux's UNFSD server [24]. The TrellisNFS server implements version 2 of the NFS protocol. We chose not to use the more recent version 3 implementation because we were not able to find a user-level implementation that had a track record of reliability in production environments.

The NFS protocol uses port 2049 by convention. The port number is in the range of ports that user-level processes are allowed to bind to; this allows our server to run as an unprivileged process.

The original UNFSD server required root privileges to access files for multiple users. At this time, the TrellisNFS server only supports a single user, thus removing the requirement that the server must run as a root process.

It should be noted that the original UNFSD server does not completely comply with NFS semantics. After a WRITE procedure call completes, data is held in the kernel's buffer cache and not immediately committed to stable storage. The consequence of this is that, if the NFS server crashes immediately after returning from a WRITE, the data will be lost and the client will falsely assume

that the operation succeeded. The TrellisNFS server semantics are the same; after a WRITE the new data is not committed to the file in the cache but is held in the kernel's buffer cache. Additionally, the write does not change the file on the home node immediately, rather the change is made after a timeout (refer to figure 3.3).

The original UNFSD server implements a data structure called the *file handle cache*. The file handle cache performs a variety of functions for the NFS server. One function we wish to identify for this section is that the file handle cache maintains a mapping of NFS file handles to server-side file paths. This cache allows the server to quickly service client requests. In the TrellisNFS server, the file handle cache maps NFS file handles to SCLs.

4.1.3 Modifications made to the original server

In order to integrate the original UNFSD server with the Trellis File System, it was necessary to make a number of changes (see Section 3.2.1). In this section, we discuss the specific implementation details of those changes.

1. *Unix API and Namespace*

   To enable the server to call the TrellisFS API functions instead of the Unix file system functions, we directly modified the server's source code. These changes were minimal since the original server localized Unix file system calls into a single file.

   Special handling of pathnames that contain embedded SCLs had to be implemented in several areas of the original server's code. The code to build a file handle from an SCL and the code to re-build an SCL from a file handle was modified. We discuss file handles in more detail later in this section.

2. *File system consistency*

   NFS supports close-to-open consistency. This means that if a file's contents are modified and the file is closed, the modifications will be visible the next time the file is opened. With the original UNFSD server, these semantics were maintained because all file operations were serialized by the operating system on the NFS server. All NFS WRITE operations were committed to the operating system kernel's buffer cache before the WRITE call is returned. Any access after the WRITE, whether it is made by an external process running on the same machine as the NFS server, or through an NFS client of the UNFSD server, will see the modified data in the kernel's buffer cache.

   With TrellisNFS, changes to a file are made immediately to the file in the Trellis cache, but changes are not reflected to the file on the home node until a call to `trellis_close()` is made. Recall that since the NFS protocol is stateless, the server has no way of determining when the application running on the NFS client closes a file. The time at which

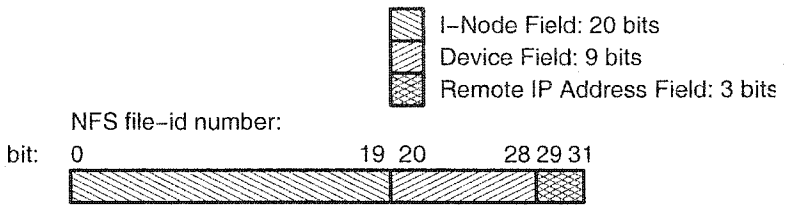


Figure 4.2: The NFS file-id numbers generated by the Trellis NFS server contain part of the original file's i-node number and device number, and the IP address of the file's home node. NFS file-id numbers are used as i-node numbers by the NFS client.

an application closes a file is the natural choice for synchronizing data between the Trellis cache and the home node. A call to `trellis_close()` is expensive, therefore calling `trellis_close()` during every WRITE procedure call is not an option.

Not knowing when a client application closes the file presents a problem for the TrellisNFS server; we solve this problem by introducing a timeout. Each file that the server has open, has a time stamp indicating when the file was last used. If the file is unused for the duration of the timeout, the TrellisNFS server will call `trellis_close()` to synchronize the file's contents with the file on the remote node. For the CISS-3 experiment, we used a timeout of one hour.

The impact this has on the Trellis environment is that after a job completes, its output data may not be visible on the home node for one hour. In a batch computing environment where a workflow takes days or weeks to complete, there is more tolerance for this timeout latency. In an environment that is more sensitive to such a large timeout latency, the length of the timeout can be reduced. We are exploring other mechanisms to detect the optimal file synchronization time.

3. *File-id number generation*

   A traditional NFS server exports files from one Unix device. The TrellisNFS server exports files from multiple home nodes (i.e., multiple servers) with multiple devices, which results in potential collisions when generating file handles. The file-id number is an important part of the NFS file handle; NFS clients use the file-id number as the i-node number.

   Files and directories in a single Unix file system are uniquely identified by their i-node and device numbers. An NFS server must map each file it serves to its own unique file-id number.

   According to the NFS protocol specification, version 2 [28], i-nodes are 32 bits in length. When generating file-id numbers, traditional NFS servers are able to use the i-node number assigned by the underlying local file system. These i-node numbers are usually 32 or 64 bits in length, as are device numbers. Network nodes on the Internet are uniquely identified by

their 32-bit IP address. Therefore, all files on the Internet can be uniquely identified by the three numbers, IP address, device number and i-node number.

Assuming 32 bit i-node and device numbers, a naive NFS server file-id generation scheme would need 96 bits to guarantee unique file-id numbers. Since the NFS protocol limits file-id numbers to 32 bits, some sort of hash function must be applied to reduce these 96 bits to only 32.

In the TrellisNFS server, we use the following scheme to generate file-id numbers. This scheme does not guarantee a file-id collision will not occur; however, the possibility of a file-id collision is rare because the scheme was designed to work in the CISS-3 experiment. We are currently working on a more general approach.

Figure 4.2 shows the breakup of the 32 available bits for the file-id number. A TrellisNFS generated i-node number has 3 bit-fields: the i-node, device and IP address bit-fields. The first 20 bits of the file-id number are the 20 least significant bits of the file's i-node number; the next 9 bits are a mask of the file's device number. It is common for Unix systems to use major and minor device numbers. We concatenate the 5 least significant bits of the minor device number with the 4 least significant bits of the major device number to obtain the 9 bits of the device bit-field of the file-id number. The last 3 bits of the file-id are for the remote node's IP address. We map an IP address to a 3 bit number, which is selected sequentially, on-demand, starting from zero. Since only 3 bits are used for the IP address part of the i-node number, the current server can serve files from at most 8 hosts at the same time. However, this number can be changed at compile time if more hosts are needed. This, of course, is done at the expense of either the accuracy of the i-node bit-field or the device bit-field.

The table that maps IP addresses to 3 bit numbers is stored in memory, and to disk. Storing this map to disk is necessary to support crash recovery in the server. If the server is restarted, the IP address map is loaded from disk so that the server is able to continue to deterministically generate file-id numbers from IP address, device number and i-node number triples.

File handle collisions are also a potential problem with the original UNFSD server. We do not solve the problem, but the potential of a file-id collision is rare, and the TrellisNFS server contains code to detect and report the occurrence of a file-id collision. We are working on a new method of generating file-id numbers that will totally avoid file-id collisions in the general case, and remove the 8-host restriction of our present scheme.

4. *The NFS File Handle*

Between client and server, NFS files are uniquely identified by an NFS file handle, which is a 32 byte opaque data structure. The NFS client does not interpret the contents of an NFS file handle. The format of a TrellisNFS file handle is shown in Figure 4.3.

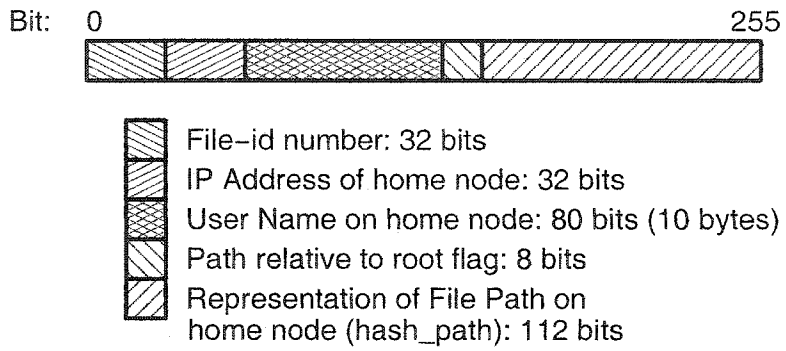


Figure 4.3: The TrellisNFS file handle contains information used to re-build an SCL from a file. An NFS file handle uniquely identifies a file between the client and the server.

The difference between a file-id number and a file handle is that the file handle contains additional information to re-build the server path of a file. In TrellisNFS, an SCL is re-built from the TrellisNFS file handle.

The content of a TrellisNFS file handle is inspired from the original user-level server (Figure 4.4 shows an example of generating a TrellisNFS file handle). A TrellisNFS file handle contains the NFS file-id number of the file, the IP address of the file's home node, the user name of the account on the home node that will be used to access the file, a boolean flag that indicates if the server path is relative to the root of the server's file system hierarchy or the users home directory, and a string of numbers known as the hash path, that helps the server determine the full pathname of the file on the remote node.

Figure 4.4 illustrates how the server builds a file handle for the SCL:

```
scp:closson@padstow:/usr/scratch/closson/water.dat
```

The file's file-id number is generated as described above and is stored in the first 32 bits of the file handle. The IP address of the file's home node is stored in the subsequent 32 bits. The next 10 bytes are used to store the name of the account to be used to access the home node; the account name is null terminated. If the account name is longer then 10 characters, the server will not be able to generate a file handle; a LOOKUP request with a user name that is longer than 10 characters will return an error. The following byte of the file handle is a boolean flag that indicates if the SCL is relative to the root of the home node's file system hierarchy or to the user's home directory. For example, the SCL `scp:padstow:water.dat` is relative to the user's home directory and the SCL `scp:padstow:/scratch` is relative to the root of the home node's file system hierarchy. For simplicity, the relative root flag occupies an entire byte instead of a single bit. The hash path is built by hashing the NFS file-ids of all the directories from the root of the SCL (either the user's home directory or the root of the remote

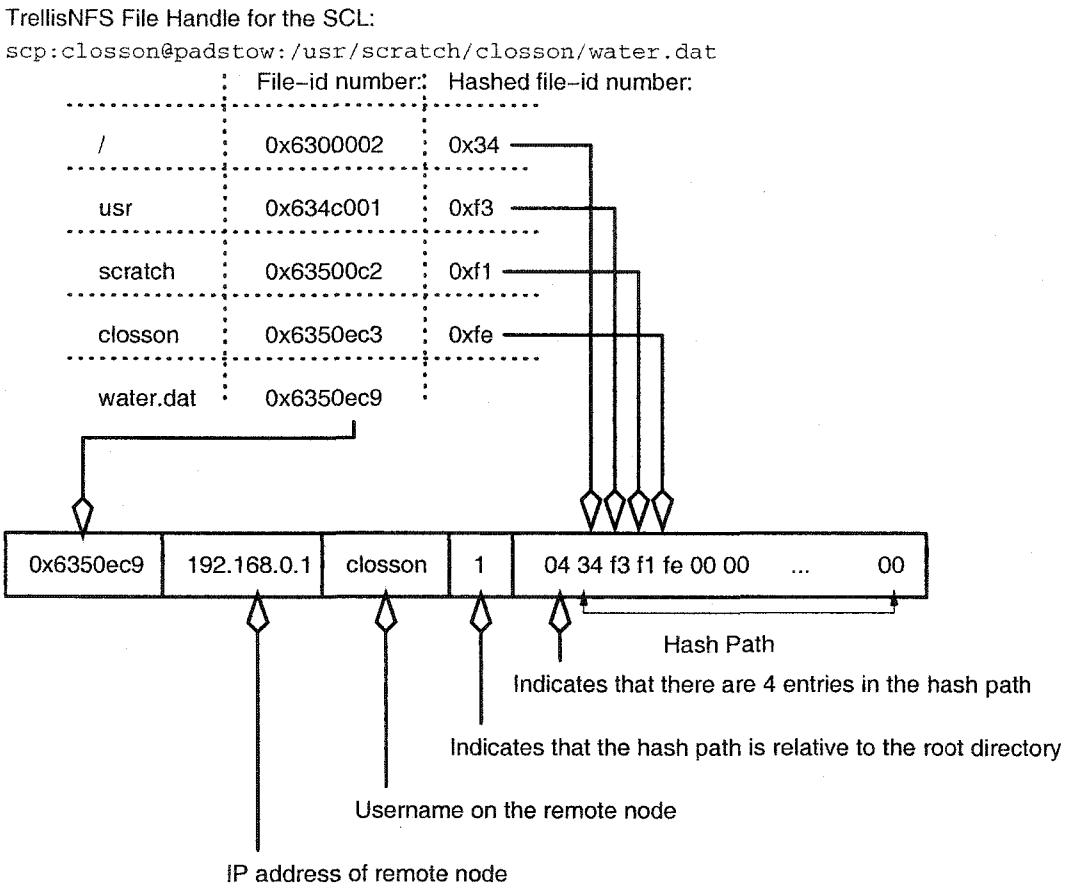


Figure 4.4: Example of how a TrellisNFS file handle is generated from an SCL.

node's file system hierarchy) to the directory containing the file in question. In Figure 4.4, the SCL is relative to the root of the remote node's file system hierarchy. The NFS file-id of the root directory is `0x6300002`. This 32 bit number is hashed to an 8 bit number, `0x34`, which is stored in the second 8 bits of the hash path. The next directory in the file's path is `usr`; its file-id number is hashed and placed in the next 8 bits of the hash path. The last entry of the hash path is the parent directory of the file. The first 8 bits of the hash path contain the length of the hash path; in this example, the length is 4. The hash path places a restriction on directory depth: files accessible with TrellisNFS can be only 13 directories deep. The original UNFSD server also has a depth restriction, but it is more then 13 levels.

One problem with this file handle format is that if a file is moved from one directory to another, the hash path changes, causing the file handle to become invalid. When our server is combined with the Linux NFS client, this problem never manifests itself because after moving a file, the NFS client will obtain the new file handle with an NFS LOOKUP remote procedure call before performing any operation on the new file handle.

If the contents of the file handle cache are lost, then the server must rebuild the SCL from the file handle. This operation involves communication with the file's home node; therefore re-building an SCL from a file handle is an expensive operation.

If a client makes a request with a file handle that is not in the server's file handle cache, the server must rebuild the SCL of the file in question. Figure 4.5 shows a detailed example of how an SCL is rebuilt from a TrellisNFS file handle.

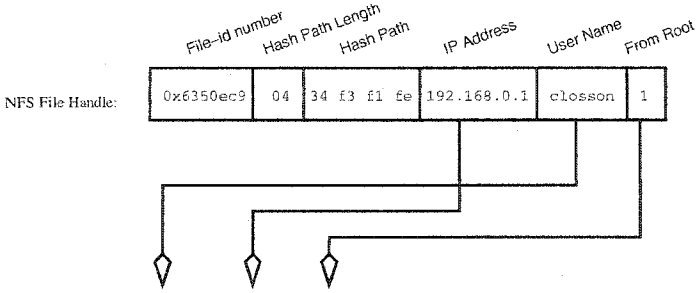
The NFS file handle contains the user name and IP address of the remote machine; it also contains the file-id number of the file on the remote machine. To locate the file, the server will perform a breadth-first search (BFS) from either the root of the remote nodes file system hierarchy, or from the user's home directory (depending on the contents of the `from_root` flag). The `hash path` component of the NFS file handle gives hints on which directory path the file resides in. The hash path effectively turns the BFS into a linear time algorithm in the average case.

4.1.4 Crash Recovery

One of the most attractive features of the NFS protocol is its simple method of crash recovery. The CISS-3 experiment ran for several months, and for long running systems, it is often the case that one part of the system will go down, and need to be restarted. The TrellisNFS server can be restarted while client programs are still running.

To provide seamless crash recovery, there were three details that needed to be taken care of:

1. *Contents of file handle cache are lost*: In the event of a server crash, the contents of the file



Initial SCL: `scp:closson@192.168.0.1:/`

The server attempts to rebuild the SCL by running a breadth first search starting from the initial SCL. The algorithm matches entries in the hash path with the hashed file-id numbers of directores encountered during the search. Once all entries in the hash path have been matched, the algorithm looks for a file with the same file-id number as the one in the file handle.

If at anytime a directory cannot be found that matches an entry in the hash path, or a file matching the file-id number in the file handle cannot be found, the algorithm terminates and returns a Stale File Handle error to the NFS client.

Steps:

- Verify that the hashed file-id of the initial SCL matches the first entry of the hash path (0x34)
- Search SCL for next entry in the hash path

lost+found	0x3d	<-- 0x3d is the hashed file-id of the lost+found directory.
boot	0xb7	
proc	0x31	
sys	0x1	
bin	0xb6	
dev	0xa8	
etc	0xf4	
lib	0xfd	
mnt	0xbb	
opt	0x7a	
tmp	0xba	
var	0xfa	
usr	0xf3	0xf3 matches the second entry of the hash path

- Descend into `scp:closson@192.168.0.1:/usr`

bin	0x32	
doc	0xb1	
lib	0x72	
man	0x70	
src	0xb7	
tmp	0xb5	
info	0xb0	
sbin	0xf7	
X11R6	0x36	
local	0xf5	
share	0x31	
libexec	0x9c	
include	0x35	
scratch	0xf1	0xf1 matches the third entry of the hash path

- Descend into `scp:closson@192.168.0.1:/usr/scratch`

test	0xa8	
closson	0xfe	0xf3 matches the fourth entry of the hash path

- Descend into `scp:closson@192.168.0.1:/usr/scratch/closson`

  We have finished searching the hash path, now search for the file-id number

water.dat	0x6350ec9

- This NFS File Handle refers to the SCL `scp:closson@192.168.0.1:/usr/scratch/closson/water.dat`

Figure 4.5: Example of how an SCL is rebuilt from an NFS file handle.

handle cache are lost. As clients issue NFS requests that contain file handles not found in the file handle cache, the server will have to re-build the SCL of the file using the method described in the previous section.

2. *The IP Address map must be committed to disk*: As mentioned in the file-id number generation section of Section 4.1.3, the server maintains a map of IP addresses. This map is used to generate file-id numbers, which become part of a file handle. After the server restarts, client requests will be processed, and the NFS file handle passed by the client will contain information from this mapping. The same mapping needs to be restored after a crash; therefore, the mapping must be committed to disk so that the same mapping can be restored when the server restarts.

3. *Synchronization of data in the Trellis cache*: The third problem is flushing data from the Trellis cache. In the event of a graceful shutdown[1], the server can flush any dirty cache data to the home node. In the event of a forced shutdown, the Trellis cache will have to be examined at start-up for any dirty data. There is enough information saved in the cache that cached and remote files can be synchronized on server startup. In the current implementation of the TrellisFS library and the TrellisNFS server, Trellis cache synchronization during a graceful shutdown has been implemented; cache synchronization in the event of a forced shutdown has not yet been implemented.

4.2 The Trellis File System Library

The Trellis File System library implements a Unix file system API [25]. The TrellisFS library is semantically equivalent to the Unix API, except that the TrellisFS library allows access to remote files, not just the files in the local file system hierarchy.

The TrellisFS library is implemented using the concept of a *wrapper function*. For example, the `trellis_open()` function *wraps* the standard Unix `open()` function. Specifically, when the `trellis_open()` function is invoked, it will first copy the remote file to the local disk, and then it will invoke the local `open()` function on this new copy.

The TrellisFS library is implemented in C++. The original implementation of TrellisFS provided functions to access and manipulate file data. We extended the library to handle file metadata, directories, directory metadata, links, and file renaming. The complete list of functions supported by the TrellisFS library, with their Unix equivalents, is provided in Table 3.1. It is our goal to make these functions as semantically equivalent to their Unix counterparts as possible. TrellisFS allows access to files on remote file systems that can be reached via the Secure Shell. Although TrellisFS provides

[1] For example, by receiving a Terminate or Interrupt signal.

access to remote files through a variety of protocols, such as SSH, HTTP and FTP, for the purposes of the TrellisNFS server we exclusively use SSH.

4.2.1 Implementation Details

TrellisFS uses whole-file caching to speed up read and write operations. Whole-file caching, as the name suggests, caches the entire file to the local disk. On a call to `trellis_open()` the whole file is fetched into the Trellis cache. On a call to `trellis_close()` the file is synchronized with its remote counterpart, if necessary. TrellisFS supports close-to-open consistency.

In addition to working with file data, the TrellisFS library also allows the user to manipulate a file's metadata. When a user queries the metadata of a file, we chose to return the metadata of the file in the cache, rather than performing a remote metadata query. We made this decision because remote communication is expensive.

Because of this policy to query the metadata of the file in the Trellis cache, and because i-node numbers must be the same during the file's lifetime, it becomes necessary to create a *placeholder* file in the Trellis cache if `trellis_stat()` is called on a file before `trellis_open()`. By creating a placeholder file, `trellis_stat()` can report an i-node number that will remain the same throughout the file's lifetime. The size and times of the placeholder file match that of the original file. The file's data is not transferred until a call to `trellis_open()` is made. Since TrellisFS does not cache directories, placeholder directories are not created.

Most TrellisFS clients do not directly use the i-node number of a file, or rely on its persistence for the client to operate correctly, however, the TrellisNFS server is an exception. I-node numbers are incorporated into the NFS file handle that is sent to the NFS client. If the i-node number of a file changes, and a client submits a request involving that file, the server may not be able to find it and will have to return a stale file handle error to the client.

Additionally, files in the Trellis cache cannot be deleted arbitrarily since the file may be used in the future. If an NFS client has an NFS file handle for a file that has been deleted from the Trellis cache, then that file handle will become stale. It is possible to replace the file in the Trellis cache with a placeholder file that preserves the i-node number, while recovering the disk space that the file once occupied.

TrellisFS functions are of two types (see Table 3.1): either the operation is performed on the file in the Trellis cache, or it is performed on the file on the home node. Operations that are performed on the remote node are referred to as remote operations. The flow of a typical remote operation in the TrellisFS library depends on whether the SSH Proxy has been enabled. Recall that the SSH Proxy (refer to Section 3.4) provides a mechanism to execute remote procedure calls over SSH.

If the SSH Proxy has not been enabled, then the TrellisFS library will copy a Perl script to the remote node and execute it. The Perl script will perform the remote file system operation and report

success or an error code. Earlier implementations of the TrellisFS library executed a Unix command to accomplish the same operation; for example, a `chmod()` operation could be performed by the `chmod` command. We chose to use the Perl script approach because it was difficult to obtain a good error code from running a Unix command. Different versions of Unix report different error messages, and rather then try to parse these human readable error messages, we return the numeric error code returned by the Perl function. The *Perl script* mechanism provides a consistent method of executing remote functions and collecting a result.

If the SSH Proxy has been enabled, then the TrellisFS library will perform the remote file system operation by connecting to a remote SSH Proxy agent through the SSH Proxy server. By using messages defined by the SSH Proxy protocol the TrellisFS library instructs the SSH Proxy agent to attempt the desired remote file system operation and return a result.

Using the SSH Proxy to perform remote file system operations leads to better performance than using the script method, which necessitates paying the price of SSH connection overhead multiple times. In addition, the SSH Proxy method does not require the overhead of forking extra processes and starting up a Perl interpreter with every operation.

4.2.2 User ID and Group ID mapping

The NFS security model relies on the NFS client to enforce access to files on a per-user granularity. An NFS client uses Unix user-id and group-id numbers to enforce security, and the NFS protocol assumes that both the client and server share a common user-id and group-id database. The TrellisNFS server is designed to work in multiple administrative domains, each with a different user-id and group-id database. To preserve the NFS security model, it is necessary for TrellisFS to maintain a map of equivalent user-id and group-id numbers between the different administrative domains.

The mapping policy is simple. Suppose that a user with a numeric uid of 100 in the local administrative domain has access to another account, with numeric uid 500, in the remote administrative domain. TrellisFS will report all files owned by user-id 500 in the remote node as being owned by user-id 100. Specifically, the `trellis_stat()` function will call the local `stat()` function on the remote node. If the uid field contains the numeric uid 500, `trellis_stat()` will replace this value with the corresponding value in the local administrative domain, 100. Files owned by all other users are reported as being owned by the special user `nobody`.

Group ID mapping is similar, except that a user may be part of multiple groups. Therefore, TrellisFS will map multiple remote group ID numbers to a single local group ID. These maps are created dynamically, on-demand, by executing the functions `getuid()` and `getgroups()` on the remote node.

4.2.3 The Metadata Cache

The TrellisFS library has an optional in-memory cache of file metadata. On a Unix file system, metadata is queried with the `stat()` and `lstat()` system calls. Figure 3.4 shows a C-style structure containing the various metadata fields.

Executing the `stat()` and `lstat()` system calls on a remote machine is expensive, especially in a high latency network. The metadata cache stores this metadata to avoid redundant remote procedure calls. The cache not only stores metadata for valid files and directories, it stores error codes if a metadata query fails.

The metadata cache is a *self-invalidated* cache. If an operation changes the metadata, the cache entry is marked invalid. The next time the metadata is queried, the TrellisFS library will have to query the remote node.

The metadata cache is optional because it weakens TrellisFS's consistency policy. If a third party externally updates the home node, say, by creating a new file, the change may not be immediately visible.

In addition to weaker consistency, when the metadata cache is enabled, hard links are not well supported. If a file is linked to two different directory entries, and one of them is unlinked, the other's i-node should be updated to show that there is now only one link. The TrellisFS library metadata cache does not currently track these relationships. Multiple links can be created and deleted, but the "number of hard links" field in the metadata of the other links will not be correct. In practise, this is a small inconsistency that does not affect the correctness of either the TrellisNFS server or of any of the applications that were run in Chapter 5.

As a rule of thumb, external updates to a remote file system will not be seen by the TrellisFS library if the metadata cache is enabled. The metadata cache does not cause a problem for the Trellis environment because the TrellisNFS server was designed for applications where all file system modifications are made through the TrellisNFS server.

The metadata cache provides a large benefit for the TrellisNFS server. NFS clients use metadata for client cache consistency and this results in frequent metadata queries. For the Connectathon basic test in Section 5.3, 30% of all NFS remote procedure calls are GETATTR calls, these calls are serviced by the metadata cache. See Section 5.3 for empirical analysis of performance gains due to the metadata cache.

4.3 Executing Remote Procedure Calls over SSH

The Secure Shell Proxy is built on top of the Secure Shell [30]. One problem faced during the implementation of the TrellisFS library is that establishing a new SSH connection costs some overhead, and by repeatedly executing TrellisFS functions that communicate over the SSH, the SSH startup

Message Name	Unix function	Description
PROXY_CHMOD	`chmod()`	Change file permissions.
PROXY_LCHOWN	`lchown()`	Change file ownership.
PROXY_LINK	`link()`	Create a new hard link.
PROXY_LSTAT	`lstat()`	Query file metadata.
PROXY_READLINK	`readlink()`	Read the value of a symbolic link.
PROXY_STAT	`stat()`	Query file metadata.
PROXY_SYMLINK	`symlink()`	Create a new symbolic link.
PROXY_UTIME	`utime()`	Update file access and modification times.
PROXY_MKDIR	`mkdir()`	Create a new directory.
PROXY_RMDIR	`rmdir()`	Delete an existing directory.
PROXY_LISTDIR	`readdir()`	Read directory contents.
PROXY_UNLINK	`unlink()`	Delete an existing file.
PROXY_RENAME	`rename()`	Rename a file.
PROXY_GETUID	`getuid()`	Query user-id number of running process.
PROXY_GETGROUPS	`getgroups()`	Query group-id list of running process.

Table 4.1: A list of the 15 different procedure calls supported by the SSH Proxy remote procedure call mechanism.

overhead quickly becomes a performance issue. To solve this problem, the designers of the original TrellisFS library also implemented the SSH Proxy as a way to maintain persistent SSH connections. The SSH Proxy provides three services to the TrellisNFS system: 1) it allows execution of arbitrary commands on a remote machine, 2) it permits copying of file data between the local and remote machine and 3) it provides a remote procedure call mechanism for executing Unix file system functions on a remote machine. We discuss only the implementation of the RPC mechanism of the SSH Proxy.

The SSH proxy system contains a mechanism to execute a set of predefined Unix file system procedure calls over the persistent connection (refer to Table 4.1 for a list of the available remote procedure calls). A diagram of the remote procedure call architecture is shown in Figure 4.6. A client that wishes to use this remote procedure call mechanism connects to the SSH Proxy server by reading and writing to a Unix domain socket. This client sends messages to the server informing it which remote node it wishes to connect to and any special connection parameters. The client can then send a predefined message indicating what procedure the client wants the remote agent to execute. For example, the client can instruct the remote agent to execute the `mkdir()` procedure call by sending a PROXY_MKDIR message and appropriate arguments. The agent will respond indicating success or an error message. All communication between the client and agent is relayed through the server.

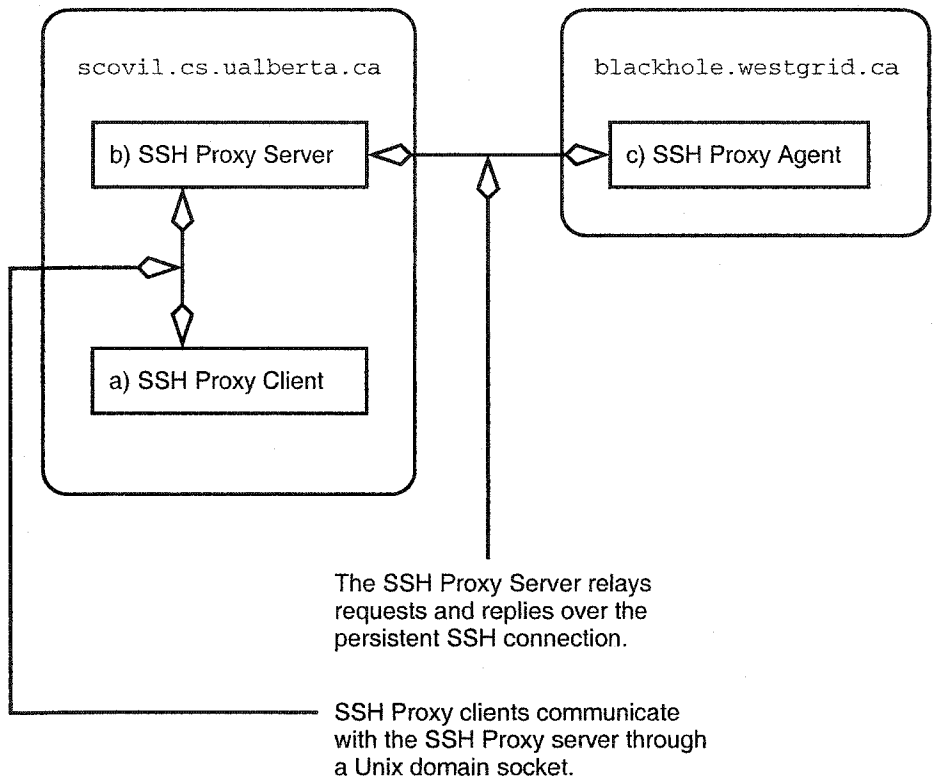


Figure 4.6: An illustration of the SSH Proxy RPC mechanism.

4.4 Concluding Remarks

In this chapter, we have discussed the implementation of the TrellisNFS server and the TrellisFS library. The TrellisNFS server owes much of its design and implementation to Linux's UNFSD server, the server that the TrellisNFS server is based on. In particular, we discussed four categories of changes that needed to be made to the server. These categories are: 1) Unix API and Namespace changes; 2) File system consistency changes; 3) NFS file-id number generation and 4) NFS File Handle generation. Additionally, we discussed changes made in the server to support the NFS crash recovery model.

We also discussed modifications made in the TrellisFS library to support full file system semantics, and not just file data manipulation. To preserve the NFS security model, we implemented functions to support all NFS operations including directory, link and meta data operations. We implemented a mechanism to automatically map user identification information between different administrative domains. Next, we discussed the Metadata cache, a cache that eliminates redundant metadata queries. Metadata is used by both the TrellisNFS server and NFS clients to determine if data in their respective internal caches is stale. We measure the performance benefits of the metadata cache in Section 5.3. Finally, we discussed the implementation of the RPC over SSH mechanism of the SSH Proxy. This mechanism allows the TrellisNFS server to perform file system operations directly on a home node.

Chapter 5

Empirical Evaluation

In this chapter, we present our evaluation of the TrellisNFS server. We evaluate the server's performance and correctness, we show that overheads introduced by integrating the TrellisFS library with the original UNFSD server are minimal. We formally verify that the TrellisNFS server and the Linux NFS client are compatible implementations, and we show the TrellisNFS server's utility by using it as part of the CISS-3 experiment.

To measure the read/write performance of the TrellisNFS server, we use the Bonnie++ [11] benchmark. For a more all-round performance evaluation of the TrellisNFS server, and to determine the interoperability of the TrellisNFS server with the Linux NFS client, we use the Connectathon test suite [27]. A valuable metric for systems software is its utility in a production environment. During the CISS-3 experiment we used two real-world applications, Gromacs [18, 6] and Charmm [8, 2], as clients of the TrellisNFS server. The results produced by these programs were used in real-world research.

5.1 Experimental Methodology and Platform

We used two micro-benchmarks to evaluate the TrellisNFS server. Our first micro-benchmark, Bonnie++, tests the raw read/write bandwidth of a file system. Sequential file reading and writing is the common case for HPC workloads. The whole-file caching strategy gives local disk performance on the TrellisNFS server. Any additional overhead when comparing the TrellisNFS server to the original UNFSD server is due to the cost of copying data to and from the home node. We use Bonnie++ to quantify the additional overhead introduced by the TrellisNFS server.

Our second micro-benchmark, the Connectathon test suite [27], has three purposes. First, it is the standard test suite to determine interoperability between an NFS client and an NFS server. We use the Connectathon test suite to determine the interoperability of the Linux NFS client and the TrellisNFS server. Secondly, we used the Connectathon test suite is used to stress test the TrellisNFS server; and third, as a benchmark. The Connectathon test suite evaluates all NFS file system

features, not just read/write performance. The TrellisNFS server has not been optimized for file system operations other than reads and writes. Therefore, we expect a relatively large performance difference between the TrellisNFS server and the original UNFSD server for file system operations other then reads and writes.

For our third evaluation method, we used the TrellisNFS server in a production environment. Figure 5.1 shows a diagram of the different test configurations used for the micro-benchmarks. We use four different test configurations for our micro-benchmarks: a) The first configuration is a single machine; file system operations are performed on the local disk. b) The second configuration is a typical NFS system; file system operations performed by an application running on the client are serviced by the original UNFSD server running on the server. c) The third configuration is the TrellisNFS system in which the home node is connected to the TrellisNFS server by a LAN; file system operations performed by an application running on the client are serviced by the TrellisNFS server running on the server. The destination of the file system operations is the home node. The local disk attached to the server is used to cache file reads and writes. d) The fourth configuration is the TrellisNFS system where the home node is connected to the TrellisNFS server by a WAN. This configuration is similar to configuration (c) in Figure 5.1 except that the home node and the TrellisNFS server are connected with a WAN. The home node is located at the University of New Brunswick and the TrellisNFS server and client are located at the University of Alberta. For all tests we used the Linux NFS client.

Network latency is a big factor in the performance of the file system; we choose whole-file caching to overcome the effects of network latency for file reads and writes. We choose these four different configurations ((a) to (c) above) to help give us an understanding of the effects of network latency on TrellisNFS.

For the Connectathon benchmarks we run two sets of benchmarks for each of configurations (c) and (d) in Figure 5.1, one set where the meta data cache is enabled, and the other where the meta data cache is not enabled. So, for the Connectathon benchmarks, a total of six configurations are used.

All local/LAN tests use the same hardware configuration. We use AMD AthlonXP processors running at 1.5 GHz, each with 1.5 GB of RAM. All local disk drives interface with the computer using a SCSI interface. The nodes were connected with a 100 Mbps switched Ethernet network. For experiments conducted on a WAN, the remote node used was located at the University of New Brunswick. Bandwidth and latency measurements for these two network configurations are shown in Table 5.1. Figure 5.2 shows the number of WAN routers an outbound TCP packet will go through between the University of Alberta and the University of New Brunswick. The latencies of each router is also shown.

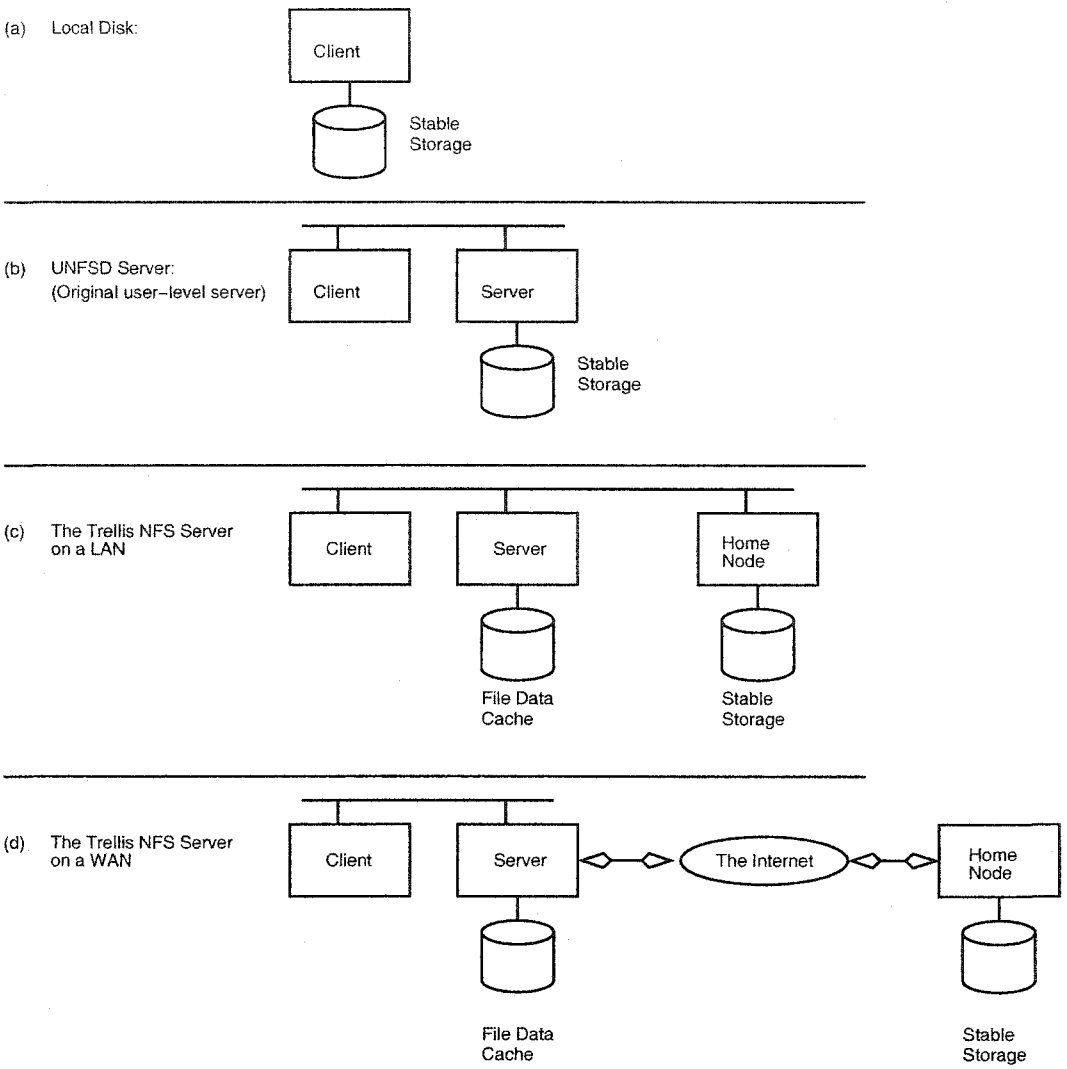


Figure 5.1: The four different test configurations used in our micro-benchmarks.

	TCP Request Response Rate (Transactions per second)	TCP Stream Bandwidth (bits/sec)
LAN	7904.32	$94.11 * 10^6$
WAN	16.80	$4.48 * 10^6$

Table 5.1: Bandwidth and latency of the networks used in our micro-benchmarks. The LAN is a 100 Mbps switched Ethernet network. The WAN connects a computer from the University of Alberta with a computer from the University of New Brunswick. These numbers were measured with the Netperf tool [16].

```
traceroute to chorus.cs.UNB.ca (131.202.139.59),
        30 hops max, 38 byte packets
 1  fawcett.cs.ualberta.ca (129.128.23.254)  1.306 ms
 2  fw-inside.cs.ualberta.ca (192.168.254.254)  0.364 ms
 3  compsci-gw.gw.ualberta.ca (129.128.153.33)  0.550 ms
 4  gsb175-netera-gsr-uofa.backbone.ualberta.ca (129.128.153.202)
        0.354 ms
 5  c4-win01.canet4.net (205.189.32.242)  17.480 ms
 6  c4-mon01.canet4.net (205.189.32.14)  49.337 ms
 7  c4-UNB.canet4.net (205.189.32.209)  59.177 ms
 8  FTNECN.ecn.UNB.ca (198.164.163.241)  59.192 ms
 9  hub-backbone.net.UNB.ca (131.202.251.201)  59.440 ms
10  131.202.139.59 (131.202.139.59)  59.211 ms
```

Figure 5.2: This figure shows the routers a packet bound for the University of New Brunswick will pass through; the latency for each router is also shown. This data was collected using the `traceroute` command.

5.2 Micro-benchmark: Bonnie++

Our first micro-benchmark is Bonnie++ [11]. The Bonnie++ benchmark tests disk throughput and CPU utilization during disk operations. In general, Bonnie++ is not used to test NFS servers or distributed file systems. However, it is still useful to compare performance between the local disk, the original UNFSD server, and the TrellisNFS server. In addition, the Bonnie++ benchmark is useful to stress test a file system.

We use the Bonnie++ benchmark to measure any overhead added on top of the original UNFSD server by its integration with TrellisFS. Our goal is to quantitatively measure that the TrellisNFS server and the UNFSD server are comparable in performance. By comparing the UNFSD server (Figure 5.1(b)) with the local disk (Figure 5.1(a)) we quantitatively measure the overhead of a user-space NFS implementation.

Bonnie++ reads and writes 3 gigabytes of data to factor out kernel buffer cache effects. We want to cancel out kernel buffer cache effects to expose the real performance of the disk and network. Any difference between the local disk and NFS configurations can then be attributed to overhead caused by the additional network traffic, and our implementation.

The Bonnie++ micro-benchmark compares the performance of the TrellisNFS server to the original UNFSD server and the local disk. Because of the different natures of these file systems, it is difficult to compare them with a single number. For example, Bonnie++ will not block while the TrellisNFS server computes MD5 hashes and performs the data transfer; neither does performing these operations consume CPU time on the client. In addition, these operations are performed on the server, freeing the client to start computing the next job. The data transfer and MD5 hash computation on the server can be performed in parallel with job execution on the client. To help capture a more realistic view of the performance of the TrellisNFS server report two sets of benchmark times:

File System Throughput (MB/s)

Configuration	Read	Write	Re-write
Local Disk	55.4	23.3	15.5
UNFSD	24.1	22.1	7.6
TrellisNFS over a LAN	23.9	22.3	7.7
TrellisNFS over a WAN	24.4	22.5	7.6

Table 5.2: NFS client performance: Bonnie++ throughput times. All results are in megabytes per second. Higher numbers are better. These numbers do not include data transfer and MD5 hash calculation.

File System Throughput (MB/s)

Configuration	Read	Write	Re-write
Local Disk	55.4	23.3	15.5
UNFSD	24.1	22.1	7.6
TrellisNFS over a LAN	7.0	5.3	3.1

Table 5.3: End-to-end performance: Bonnie++ throughput times. All results are in megabytes per second. Higher numbers are better. These numbers include data transfer and MD5 hash calculation.

one will not include data transfer and MD5 hash computation times, and the other will.

5.2.1 Test Description

There are 3 stages in the Bonnie++ benchmark: write, read, and re-write. First, three 1 gigabyte files are created and written using the `write()` system call. Second, the 3 gigabytes of data is read back using the `read()` system call. Third, the 3 gigabytes of data is split into 16 KB pages; each page is read, dirtied and re-written, which requires a call to `lseek()`. Each of the 3 tests was performed 10 times; results are an average of these 10 runs.

As mentioned in the previous section, we report two sets of benchmark times; one set contains MD5 hash computation and data transfer times, the other set does not. In order to include these additional overheads we modified the Bonnie++ benchmark program to instruct the TrellisNFS server to synchronize the Trellis cache after each of the three benchmark phases.

The benchmark set that involves measuring data copying overheads and MD5 checksum computation would result in copying an unreasonable amount of data over the Internet, therefore we decided not to run the benchmark with this configuration. Since the cost of copying data over the Internet is much more expensive then copying data over a LAN, we expect the benchmark times to be much lower for the TrellisNFS over a WAN configuration ((d) in Figure 5.1).

5.2.2 Results

Figure 5.4 and Table 5.3 show the throughput of the read, write and re-write tests, including the additional MD5 computation and data transfer overheads. Figure 5.3 and Table 5.2 contain results

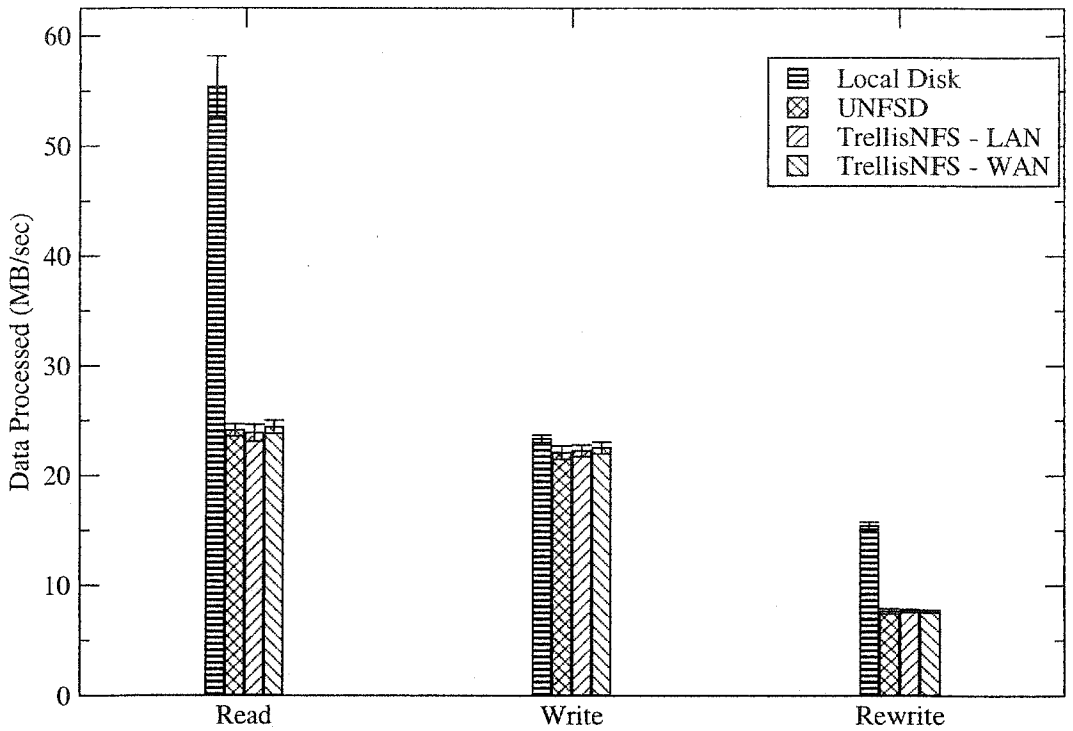


Figure 5.3: NFS client performance: Bonnie++ throughput times. All results are in megabytes per second. Higher numbers are better. These numbers do not include data transfer and MD5 hash calculation.

CPU Utilization (%)

Configuration	Read	Write	Re-write
Local Disk	21.9	21.7	9.4
UNFSD	10.1	21.2	6.4
TrellisNFS over a LAN	9.9	21	6.1
TrellisNFS over a WAN	10.1	21.3	6.2

Table 5.4: NFS client performance: Bonnie++ CPU utilization. Numbers are percentages. Lower numbers are better. These numbers do not include data transfer and MD5 hash calculation.

CPU Utilization (%)

Configuration	Read	Write	Re-write
Local Disk	21.9	21.7	9.4
UNFSD	10.1	21.2	6.4
TrellisNFS over a LAN	2.8	4.7	2.6

Table 5.5: End-to-end performance: Bonnie++ CPU utilization. Numbers are percentages. Lower numbers are better. These numbers include data transfer and MD5 hash calculation.

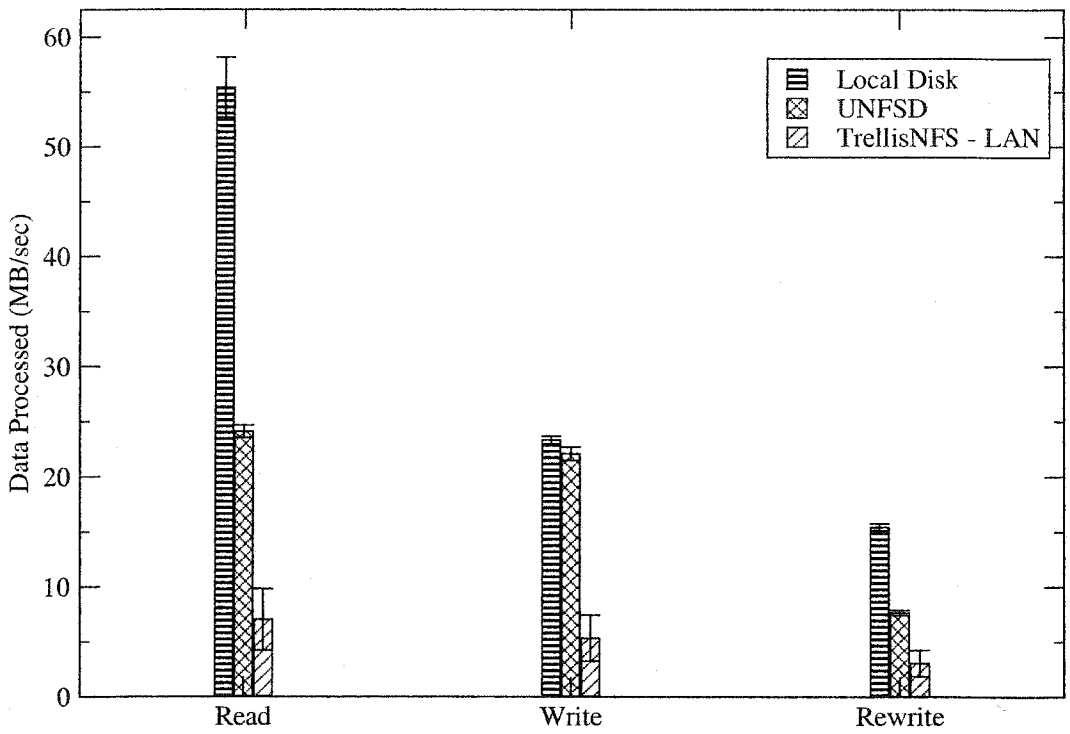


Figure 5.4: End-to-end performance: Bonnie++ throughput times. All results are in megabytes per second. Higher numbers are better. These numbers include data transfer and MD5 hash calculation.

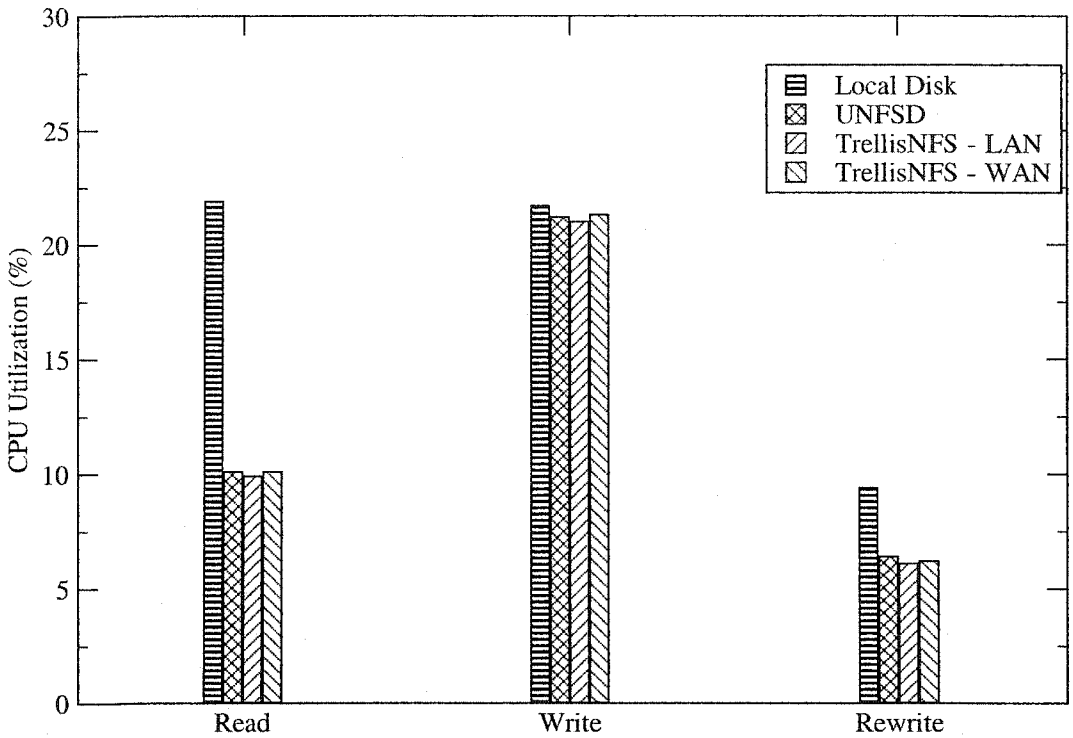


Figure 5.5: NFS client performance: Bonnie++ CPU utilization. Number are percentages. Lower numbers are better. These numbers do not include data transfer and MD5 hash calculation.

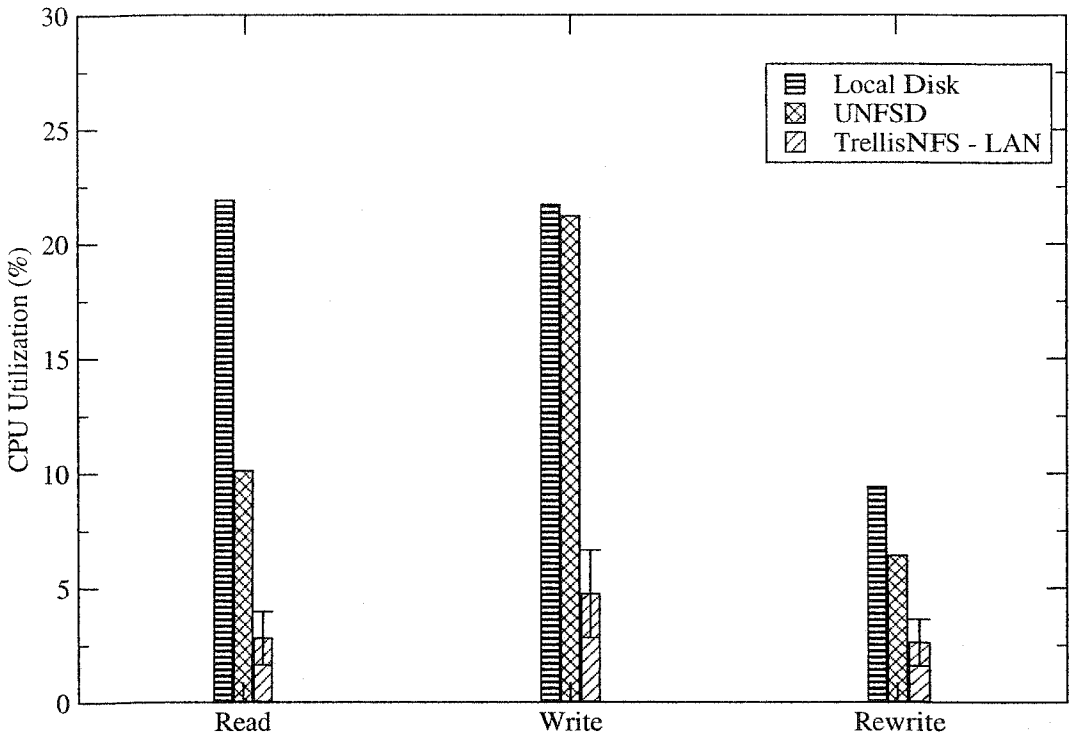


Figure 5.6: End-to-end performance: Bonnie++ CPU utilization. Number are percentages. Lower numbers are better. These numbers include data transfer and MD5 hash calculation.

that do not include these additional overheads. The read test, without accounting for MD5 hash computation and data transfer overheads, show virtually identical performance between the three NFS server configurations (Figure 5.1 (b), (c), and (d)). Local disk read performance is about 2.3 times faster then the NFS configurations. The write test shows all four test configurations have almost equal performance. The NFS configurations perform about 1 MB/s slower then local disk. The re-write test shows a similar pattern as the read test; which is, the throughput of the local disk configuration is about twice that of the NFS configurations, and the three NFS configurations have about the same performance. The read test that does include MD5 hash computation and data transfer overheads shows the original UNFSD server's read bandwidth is about 3.4 times faster then that of the TrellisNFS server. The write and re-write tests show a performance disparity of 4.2 and 2.5 times respectively.

File operations performed on the NFS configurations require a network message to be sent to the server, the disk operation to be performed, and a reply to be sent back to the client. For local disk operations, only the disk operation is performed, no network communication takes place. We attribute the difference in performance between the local disk and the NFS configurations to this network traffic, and the synchronous nature of NFS operations. Although the average throughput of TrellisNFS over a WAN is higher in some tests than with the original UNFSD server, we attribute this to benchmark noise since there is significant overlap in the error bars shown on Figure 5.3.

For the TrellisNFS configurations, a file is created on the remote node, but all reads and writes are processed out of the Trellis cache. The overhead of creating this remote file is amortized by the length of time spent writing or reading from disk. The Bonnie++ benchmark only measures file system performance from the perspective of the client, and since MD5 hash computation and data transfer overheads take place on the server, they are not measured by the Bonnie++ benchmark. To get some understanding of the impact of these additional overheads we chose to modify the Bonnie++ benchmark; our modification makes update of file data between the TrellisNFS server and the home node, synchronous. This modification will account for the overheads not measured by unmodified Bonnie++, but still does not truly reflect the performance of the TrellisNFS server as Trellis cache synchronization can be done in parallel with processing on the NFS client. Never-the-less, both measurements are valuable to gain an understanding of the performance of the TrellisNFS server.

Figure 5.6 and Table 5.5 show CPU utilization during the tests that include MD5 hash computation and data transfer overheads. Figure 5.5 and Table 5.4 show CPU utilization during the tests that do not include MD5 hash computation and data transfer overheads. The CPU utilization tests show a similar trend to the throughput tests, except that the NFS server configurations consume less CPU time then the local disk. The local disk read test differs from the NFS configurations by a factor of about 2. We attribute this to the fact that the actual disk operations are performed on the NFS server,

whose CPU utilization was not measured. CPU utilization during the write test is about equal for all four configurations. The re-write test also shows a similar trend to the read test, although not as exaggerated.

The CPU utilization test that measures MD5 hash computation and data transfer overheads shows even less CPU usage then the test that does not include these overheads. This is because these operations are performed on the TrellisNFS server, leaving the client CPU idle during these operations.

5.2.3 Conclusion

For the purposes of the Bonnie++ benchmark, the primary difference between the original UNFSD server and the TrellisNFS server is that the TrellisNFS server will create a file on the home node at the beginning of the benchmark and potentially copy data between the home node and the TrellisNFS server at the end of each phase. It is important to measure the effects of these differences, but designing a method of doing so is difficult. We chose to report two sets of results; one set does not include MD5 hash computation and data transfer overheads; this set is more indicative of NFS client performance. The other set of results does include these overheads, this set is more indicative of end-to-end system performance.

The results show that the TrellisNFS server adds minimal overhead to the original UNFSD server when performing reads and writes out of the Trellis cache; and that the cost of computing MD5 hashes and data transfer is significant. By comparing the original UNFSD server and the TrellisNFS server to the local disk, we see that disk performance, and not network performance, is the bottleneck to write throughput.

5.3 Micro-benchmark: The Connectathon NFS Test Suite

The Connectathon NFS [27] test suite is the *de facto* test suite to ensure NFS client and server compatibility. Connectathon is a yearly event sponsored by Sun Microsystems and other industrial and academic institutions. NFS implementors are invited to test their individual client and server implementations with each other. We used the test suite from the 2003 Connectathon event.

We used the Connectathon suite for three purposes: 1) To test compatibility between the TrellisNFS server and the Linux NFS client. According to the Connectathon suite, TrellisNFS and the Linux NFS client are 100% compatible. 2) As a stress test, to evaluate the server's correctness and stability while operating under a heavy load. And 3) to use the Connectathon tests as a benchmark. Using the Connectathon suite as a benchmark will provide a more all-round evaluation than that provided by the Bonnie++ benchmark. We have not optimized non-read/write operations and we expect the performance of these operations to be much worse then that of the original server or the local disk. Also, we measure the performance benefit of the metadata cache with the Connectathon test suite.

5.3.1 Test Description

The Connectathon test suite consists of four tests: basic, general, special and locking. Each test consists of running several programs on an NFS client that mounts an exported volume from an NFS server.

The basic test is designed to test individual NFS procedure calls. Unix file system API calls are used to exercise the file system. The general test exercises the server by running a series of standard Unix programs. The general test simulates an interactive user session.

We use two of the four Connectathon tests to evaluate the correctness of the TrellisNFS server. The special and locking tests are not performed, since neither the TrellisNFS server nor the original unmodified UNFSD server pass them. The Connectathon test suite documentation explicitly states that the special and locking tests are optional. A client and server pair is considered 100% interoperable if they pass the basic and general tests.

We used Linux 2.4.18 as our NFS client. This pair (the TrellisNFS server and Linux client) pass both the basic and general tests. The only other client we tested with the TrellisNFS server was the NFS client in the Irix operating system. This pair did not pass either the basic or the general tests. Since we only planned to use Linux clients for evaluating the server, we did not investigate the cause of the failure.

5.3.2 Benchmark setup.

When using the Connectathon test suite as a benchmark, we test the four configurations shown in Figure 5.1. The configurations are: a) local disk; b) the original UNFSD server; c) the TrellisNFS server on a LAN, with and without enabling the metadata cache; and d) the TrellisNFS server on a WAN, with and without enabling the metadata cache. We ran both the basic and general Connectathon tests in each configuration. Results from each test were gathered by averaging 10 runs.

Running Connectathon on the local disk is done purely for baseline comparison. Local disk benefits from memory caching for writing data and metadata, as provided by the operating system. The NFS protocol explicitly states all writes must be synchronous. In contrast, a local disk system does not perform synchronous writes.

There are additional overheads in the TrellisNFS server when compared to the original UNFSD server. With the TrellisNFS server configurations, (Figure 5.1(c) and (d)) data must be copied back and forth between the home node and the TrellisNFS server. In contrast, in the UNFSD server configuration (Figure 5.1(a)), the final location for the data is on the server's local disk. In the TrellisNFS server, all operations other than reads and writes are synchronous across both the TrellisNFS server and the home node. To further illustrate the difference between a synchronous NFS operation and a synchronous TrellisNFS operation, refer to Figures 5.7 and 5.8. For non-read/write operations, the client blocks until network messages are sent to the TrellisNFS server, then to the home node,

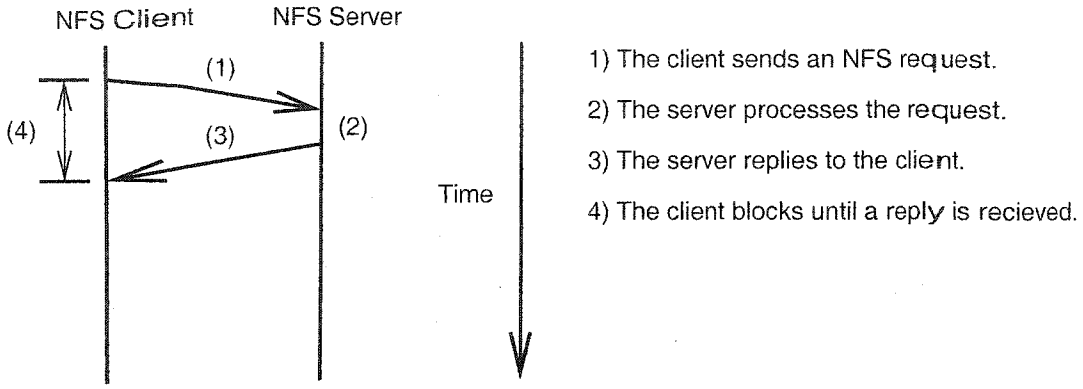


Figure 5.7: In a typical NFS synchronous operation, the client blocks while the request is processed on the server.

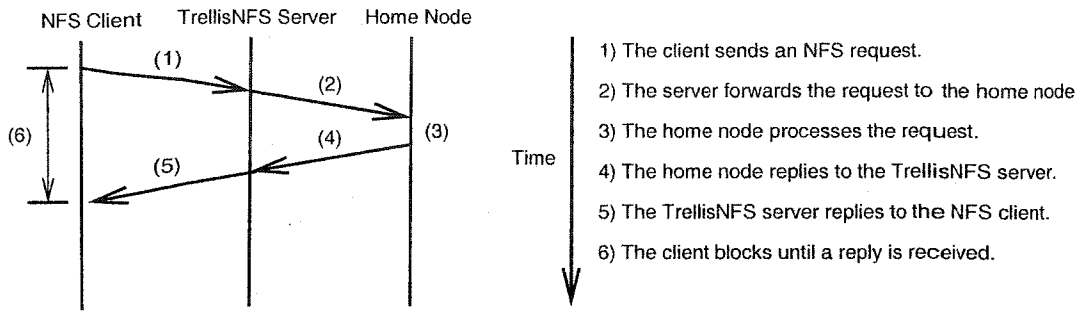


Figure 5.8: In a typical TrellisNFS synchronous operation, the client blocks while the request is processed on the home node. Read and write operations in the TrellisNFS server are not processed on the home node, but on the TrellisNFS server.

back to the TrellisNFS server and, finally, back to the NFS client.

When testing the TrellisNFS server on a LAN, the remote node is attached to the same LAN as the server and client. To test the TrellisNFS server on a WAN, we use a machine located at the University of New Brunswick. A comparison of network bandwidth and latency between our LAN and WAN configurations is shown in Table 5.1.

5.3.3 Results

The primary purpose of using the Connectathon suite in this evaluation was to determine compatibility between the TrellisNFS server and the Linux NFS client. By passing the basic and general tests, the Connectathon test suite certifies that our server and the Linux NFS client are 100% compatible with each other.

The second purpose of the Connectathon suite was to stress test the TrellisNFS server. In order to do this, we ran two instances of both the basic and general test in a continuous loop. The basic test is a file system intensive test; running a single instance of the basic test produces enough NFS

traffic to keep the TrellisNFS server operating full-time. We also ran two instances of one of the CISS-3 applications in a continuous loop. All six of these clients ran continuously for the period of two weeks, and in this time the server continued to operate without degraded performance.

The third purpose of the Connectathon test suite was to use it as a benchmark, which gives us a more complete view of the TrellisNFS server's performance. Unlike the Bonnie++ benchmark, the Connectathon test suite exercises all file system functionality, not just file reads and writes. The results of the benchmark are shown in Figures 5.9, 5.10, 5.11 and 5.12. Also, the results are shown in table form in Tables 5.6, 5.7 and 5.8. We attribute the difference in the performance of the TrellisNFS and the original UNFSD server to three causes: 1) The additional network overhead of performing synchronous remote procedure calls between three computers instead of only two (see Figures 5.7 and 5.8); 2) the overhead of encrypting data with the SSH [12] and 3) not optimizing our implementation for non-read/write operations. By using the Connectathon test suite as a benchmark, we also quantified the performance benefit of the metadata cache.

For each test in the basic and general test suites, the test programs go through the same administrative startup work. Each test checks for and, if necessary, removes the old test directory; a new directory to perform the tests is then created.

For each test set, we started with a clean directory, so no unnecessary cleanup was required, that might skew results. The creation of the test directory requires a handful of remote operations that will affect all tests, even those that do not perform any remote operations.

1. The cumulative times for the basic test are shown in Figure 5.9 and Table 5.6. In the best case, the TrellisNFS server is six times slower then the original UNFSD server. This is because: 1) all operations are performed on the home node; and 2) there is additional overhead due to the extra network communication between the TrellisNFS server and the home node. The illustration in Figure 5.8 shows that for a single NFS operation between the NFS client and the TrellisNFS server, there is only one remote operation between the TrellisNFS server and the home node; but in most cases, there are several remote procedure calls between the TrellisNFS server and the home node. Often the original UNFSD server will perform redundant calls to `stat()`. Performing these extra calls is not a problem for the original UNFSD server because the `stat()` function is very fast when performed on a local disk. In the TrellisNFS server, `stat()` calls result in communication with the home node. The performance benefit of the metadata cache comes from eliminating these multiple redundant `stat()` calls. And finally, 3) all communication between the TrellisNFS server and the home node bears the additional overhead of SSH encryption.

2. Details for each phase of the basic test are shown in Figure 5.11 and Table 5.7. The STAT ROOT, READ WRITE and STATFS tests stand out because even in the worst case they com-

	Basic Test	General Test
Local Disk	0.37	3.05
UNFSD	2.46	3.63
LAN w/ Metadata Cache	12.27	5.55
LAN w/o Metadata Cache	27.99	7.94
WAN w/ Metadata Cache	369.08	48.79
WAN w/o Metadata Cache	1080.84	259.59

Table 5.6: Execution times for the Basic and General Connectathon Test Suites. Times are in Seconds.

plete an order of magnitude faster than the other tests. These three tests result in much less communication between the TrellisNFS server and the home node, and this is the nature of the performance improvement.

3. The cumulative times for the general test are shown in Figure 5.10 and Table 5.6. There is less disparity between different configurations (Figure 5.1) with the general test than with the basic test. This difference can be attributed to the fact that the general test is less I/O bound than the basic test.

4. Details for each phase of the general test are shown in Figure 5.12 and Table 5.8. Performance differences among the different configurations can be attributed to the same three reasons identified above in point number 1. The performance gains due to the metadata cache are most noticeable in the Makefile test. Makefiles use file timestamps to determine if build dependencies need to be run. Querying timestamps results in a call to `stat()` in the TrellisNFS server. In the best case, the metadata cache improves the performance of the Makefile test 7-fold, because the metadata cache eliminates redundant `stat()` calls.

5.3.4 Conclusion

The purpose of using the Connectathon NFS test suite to evaluate the TrellisNFS server is threefold: 1) To certify that the TrellisNFS server is 100% compatible with the Linux NFS client; we have achieved this goal; 2) To stress test the TrellisNFS server. We found that the TrellisNFS server ran reliably in a demanding environment for the two-week period; and 3) as a benchmark to get a more all-round evaluation of the TrellisNFS server. We attribute the difference in performance of our three configurations to three causes: the cost of performing synchronous remote procedure calls between three computers, instead of two, the additional overhead of encrypting data with the SSH [12], and our unoptimized implementation of non-read/write operations. We also quantified the performance benefit of the metadata cache.

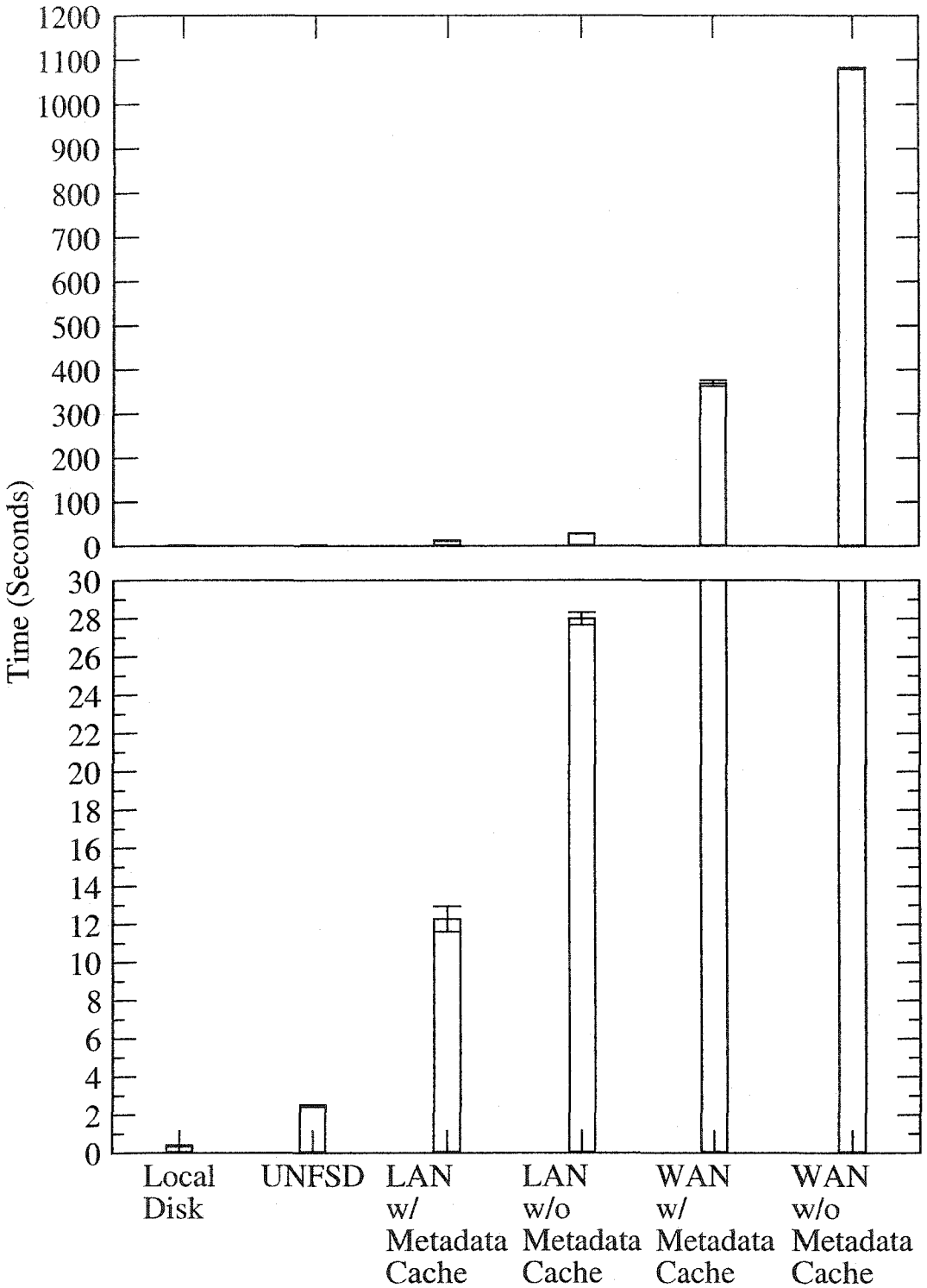


Figure 5.9: Completion times for the Basic Connectathon Test Suite. The top plot shows the performance of all configurations. The bottom plot focuses on the completion times of the first four configurations.

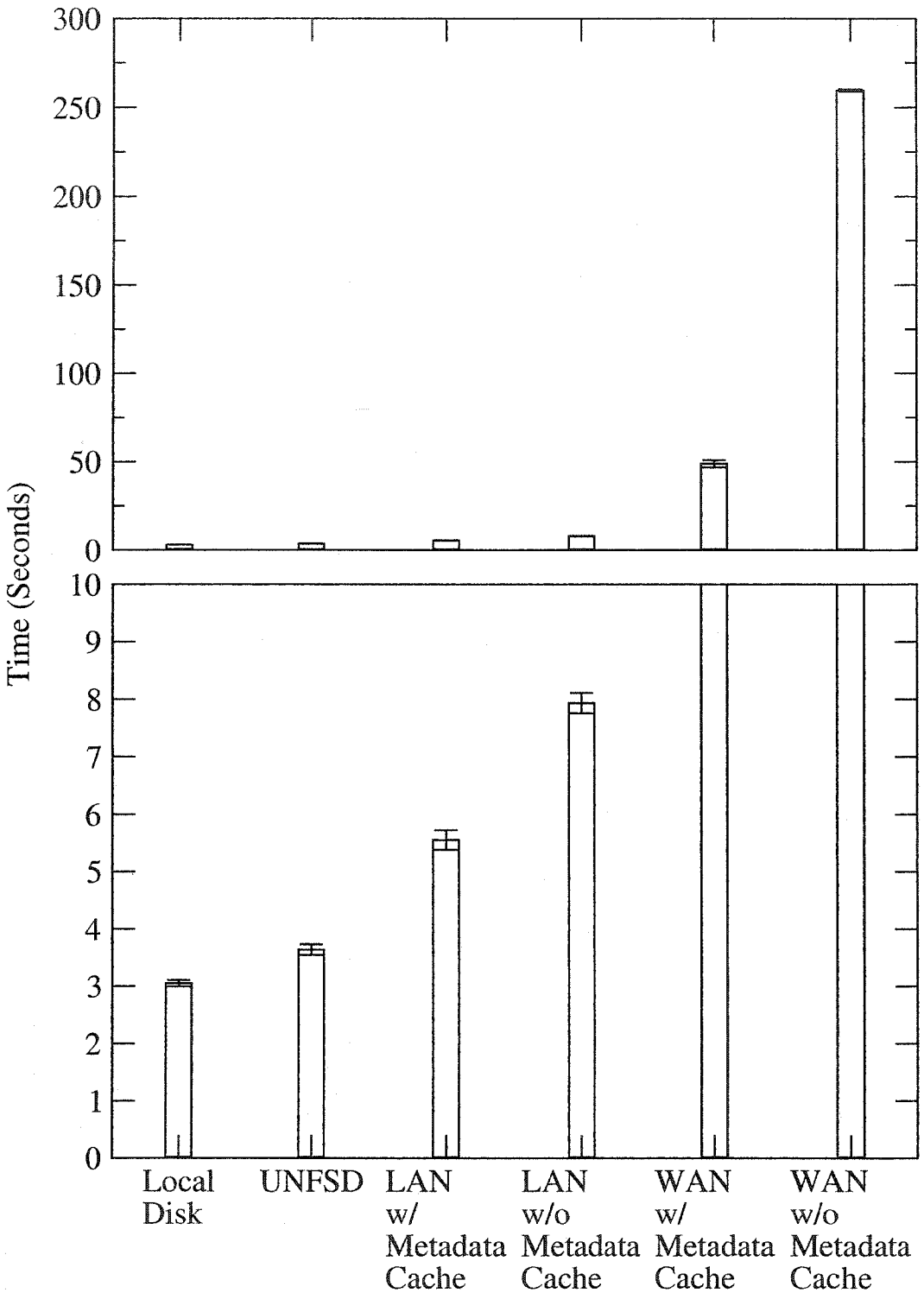


Figure 5.10: Completion times for the General Connectathon Test Suite. The top plot shows the performance of all configurations. The bottom plot focuses on the completion times of the first four configurations.

	MKDIR CREATE	RMDIR UNLINK	STAT ROOT	CHMOD *ATTR	READ WRITE
Local Disk	0.03	0.01	0.02	0.02	0.12
UNFSD	0.15	0.15	0.01	0.41	0.33
LAN w/ Metadata Cache	3.01	0.32	0.02	1.28	0.36
LAN w/o Metadata Cache	4.78	1.53	0.05	2.98	0.46
WAN w/ Metadata Cache	65.44	15.63	0.72	68.62	1.99
WAN w/o Metadata Cache	118.34	73.59	3.87	163.80	8.05

	READDIR	RENAME	SYMLINK READLINK	STATFS
Local Disk	0.03	0.01	0.02	0.06
UNFSD	0.45	0.19	0.30	0.44
LAN w/ Metadata Cache	4.31	1.68	0.83	0.40
LAN w/o Metadata Cache	9.08	4.21	4.30	0.44
WAN w/ Metadata Cache	103.61	45.44	66.22	0.85
WAN w/o Metadata Cache	279.30	167.77	260.36	3.87

Table 5.7: Execution times of selected phases of Connectathon's Basic Test. A plot of these times is shown in Figure 5.11

	Small Compile	Tbl	Nroff	Large Compile	(x4) Large Compile	Makefile
Local Disk	0.49	0.06	0.24	0.63	1.38	0.16
UNFSD	0.58	0.08	0.26	0.73	1.50	0.37
LAN w/ Metadata Cache	0.68	0.22	0.30	0.85	1.66	1.48
LAN w/o Metadata Cache	0.78	0.28	0.37	0.94	1.94	2.99
WAN w/ Metadata Cache	3.19	1.90	1.06	3.39	10.81	21.30
WAN w/o Metadata Cache	11.87	6.54	9.40	11.69	40.14	159.04

Table 5.8: Execution times of selected phases of Connectathon's General Test. A plot of these results is shown in Table 5.12

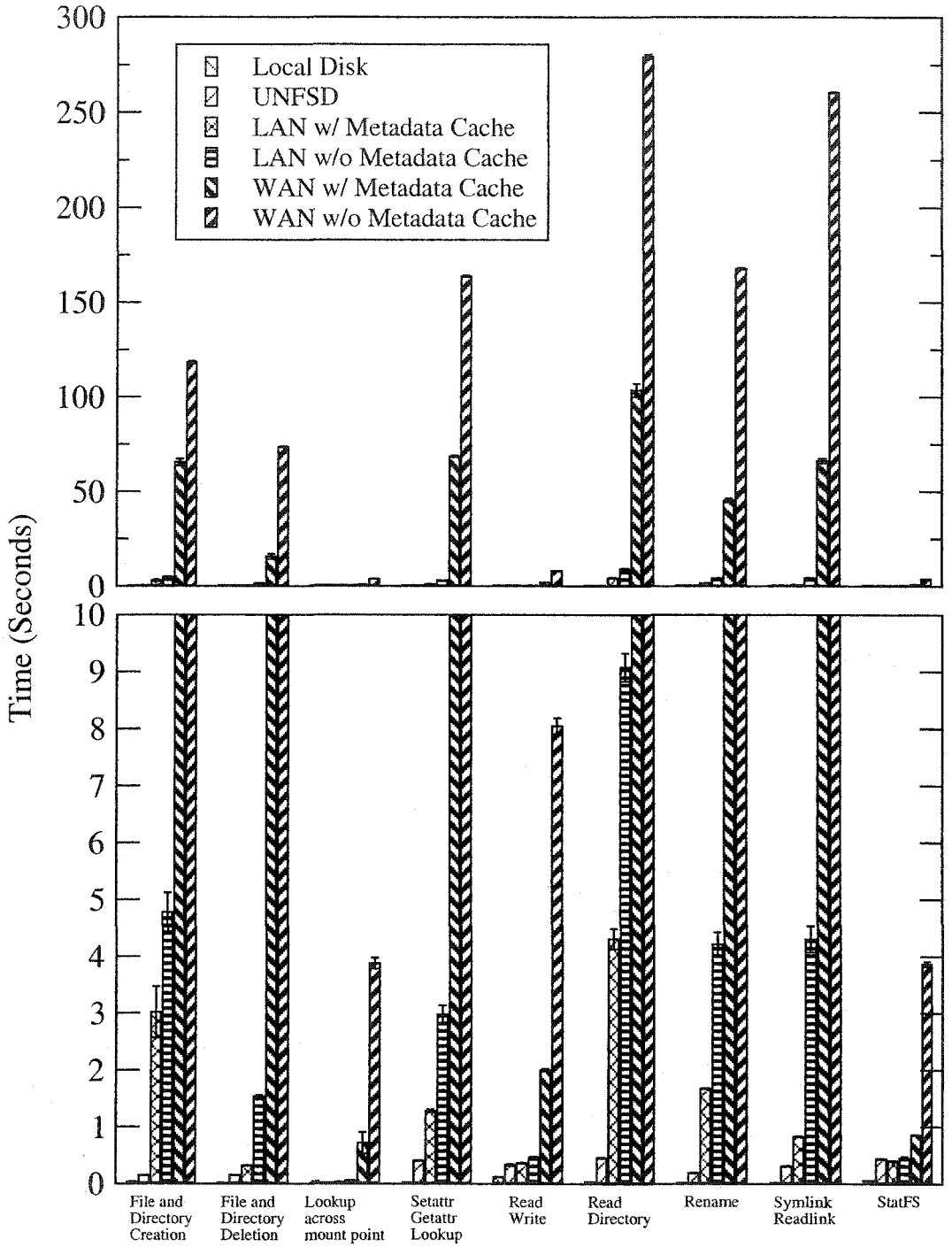


Figure 5.11: Completion times for the individual phases of the Basic test of the Connectathon Test Suite. The top plot shows the performance of all configurations. The bottom plot focuses on the completion times of the first four configurations. Table 5.7 shows these results in table format.

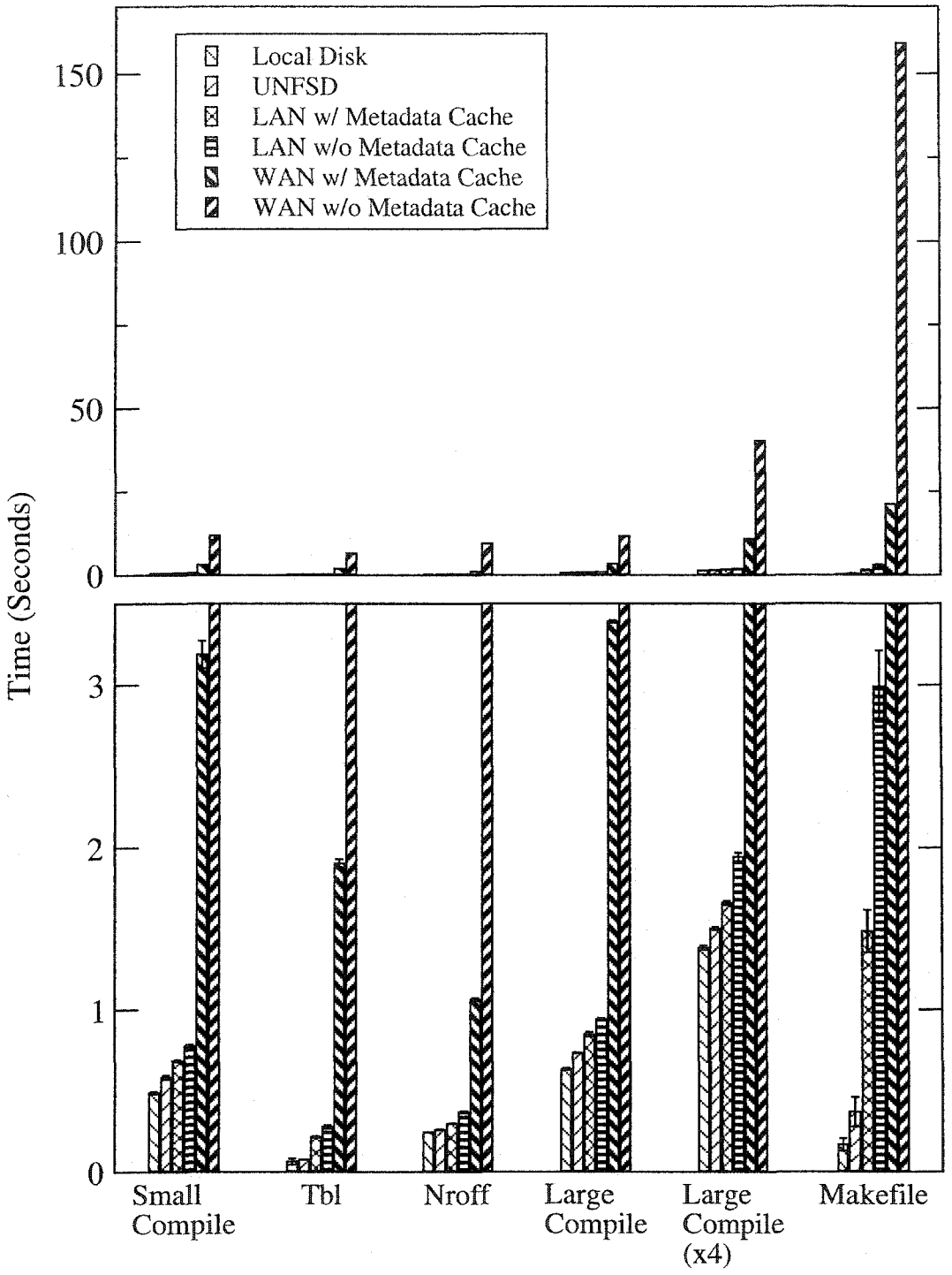


Figure 5.12: Completion times for the individual phases of the General Test of the Connectathon Test Suite. The top plot shows the performance of all configurations. The bottom plot focuses on the completion times of the first four configurations. Table 5.8 shows these results in table format.

5.4 Application-Oriented Benchmark: The Third Canadian Internetworked Scientific Supercomputer

We used the TrellisNFS server as part of the CISS-3 Metacomputer. The CISS-3 experiment is a production environment, and results from the CISS-3 experiment will be incorporated into real-world research results. The primary benefit of using the TrellisNFS server in the CISS-3 experiment is to show the utility of the server in a real research environment; a secondary benefit is that we can further evaluate the quality of our implementation. As of August 2004, the CISS-3 experiment has now run for 3 months. The CISS-3 experiment will run through the month of September 2004.

During the CISS-3 experiment, we are running two applications. The first application, Gromacs [18, 6], is a molecular dynamics simulator written in C, with in-line x86 assemble code. The second application, Charmm [8, 2], is a macromolecular simulator written in Fortran.

We now describe the architecture of the CISS-3 Metacomputer, focusing on the parts most relevant to the TrellisNFS server. The CISS-3 Metacomputer encompassed research organizations from all across Canada; every province, except P.E.I., was represented. The TrellisNFS server was used in two of the many administrative domains participating in the CISS-3 experiment. The first site to use the TrellisNFS server was the Jasper cluster at the University of Alberta; this is a 20-node, 40-processor Linux cluster. The second site to use the TrellisNFS server was the Chorus cluster at the University of New Brunswick; which is an 80-node, 160-processor Linux cluster. There are two data storage servers in the CISS-3 Metacomputer; the input and output data for the Gromacs application is stored on the server `squirrel.bio.ucalgary.ca` located at the University of Calgary. The input and output data for the Charmm application is stored on the server `blackhole.westgrid.ca` at Simon Fraser University. Both of the TrellisNFS servers provided seamless access to the file systems on the two remote data storage servers. As of September 2, 2004; the CISS-3 experiment had been running for 20 weeks.

5.5 Concluding Remarks

In this chapter, we have measured the performance of the TrellisNFS server. We used three methods to evaluate our implementation: we used the Bonnie++ micro-benchmark to test the read/write performance of the server; we used the Connectathon test suite to measure the compatibility, stability, and performance or the server; and finally, we used the TrellisNFS server in the CISS-3 experiment, a production environment.

Through the Bonnie++ benchmark we see that from the perspective of the NFS client, the TrellisNFS server has equal performance to the original UNFSD server. We also measured end-to-end system performance to determine the cost of copying data over the network and computing MD5 hashes; we saw that these costs are significant.

The Connectathon benchmark was used to certify the compatibility of the TrellisNFS server with the Linux NFS client. Additionally, we used the Connectathon test suite, with other programs, to stress test our implementation. Through this exercise, we found our implementation works reliably under heavy load. The third purpose of the Connectathon test suite was to help us see a more all-round evaluation of the performance of the TrellisNFS server. The Bonnie++ benchmark only measures the performance of file system reads and writes; the Connectathon test suite exercises all file system functionality. Our implementation does not perform as well as the original UNFSD server; we identified three sources of additional overhead in the TrellisNFS server that do not affect the original UNFSD server.

As an additional measure of the utility of our design and the stability and usefulness of our implementation we used the TrellisNFS server as a shared file system in the CISS-3 metacomputer. The CISS-3 experiment began April 15, 2004 and will continue through the month of September 2004. The TrellisNFS server has performed well in the CISS-3 environment.

Chapter 6

Conclusion

In this work, we presented the design and implementation of the TrellisNFS server. A key concept in the design of TrellisNFS is that the TrellisNFS server allows unmodified binaries to work with remote files. Another key design concept is preserving compatibility with existing NFS clients; to ensure compatibility, we do not modify the NFS protocol or change semantics that existing NFS clients expect from the original UNFSD server.

We modified Linux's UNFSD server and integrated it with the TrellisFS library. We used aggressive caching to deal with the high latencies of wide area networks. The TrellisNFS server implements last-writer-wins consistency semantics. This design choice eliminates expensive WAN communication that would be needed to support stronger consistency semantics; our motivating applications do not require stronger consistency semantics. Implementing the TrellisNFS server required a re-design of key NFS data structures to allow the server to work with files from multiple servers. We also took care to preserve the NFS model of crash recovery.

We expanded the scope of the TrellisFS library to allow it to work with more file system features; such as directories, links, file renaming and metadata. We have implemented a metadata cache, which is designed as a general purpose mechanism to eliminate redundant calls to the `stat()` and `lstat()` system calls.

We implemented a mechanism to execute remote procedure calls over a persistent SSH connection, and we use this mechanism extensively to implement file system functionality in the TrellisFS library.

We have evaluated the TrellisNFS server using three methods. We used the Bonnie++ benchmark to evaluate the performance of individual read and write operations. Read and write operations are the common case for HPC applications, and because of this we have focused on them with this benchmark. From the perspective of the NFS client, read/write performance is equivalent to that of the original UNFSD server. We observed that performance differences between the UNFSD server and the TrellisNFS server can be attributed to computing MD5 checksums and data copying over the network. We used the Connectathon NFS test suite for three purposes: 1) to determine inter-

operability between the Linux NFS client and our TrellisNFS server; 2) as a stress test to evaluate performance and stability of the TrellisNFS server while under heavy load; and 3) as a benchmark to get a more all-round evaluation of TrellisFS' performance than that of Bonnie++. We determined that the TrellisNFS server and the Linux NFS client are 100% compatible and that the TrellisNFS server is stable under high load. Performance of non-read/write operations are more expensive under the TrellisNFS server than under the original UNFSD server. We attribute this to three factors: 1) the cost of performing synchronous remote procedure calls between three computers instead of two; 2) the additional overhead of encrypting data with the SSH; and 3) our unoptimized implementation of non-read/write operations.

We have used the TrellisNFS server in a production environment, producing real research results. Through the CISS-3 experiment we have shown the utility of our design and the stability of our implementation.

In a Trellis metacomputer, compute jobs can be assigned to any node. The TrellisNFS server allows files from any storage node to be accessed by any compute node. Also, application programs do not have to be modified to take advantage of the TrellisNFS distributed file system.

Bibliography

[1] *The Linux Manual: section 2, stat() and lstat()*, 1998.

[2] A. D. MacKerell, Jr., B. Brooks, C. L. Brooks, III, L. Nilsson, B. Roux, Y. Won, and M. Karplus. CHARMM: The Energy Function and Its Parameterization with an Overview of the Program. *The Encyclopedia of Computational Chemistry*, 1:271–277, 1998.

[3] A. Tridgell and the Samba Team. The Samba Website. http://www.samba.org.

[4] 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*, 16(3):207–233, 1998.

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

[6] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Comp. Phys. Comm.*, 91:43–56, 1995).

[7] J. C. Bowman. Secure nfs. http://www.math.ualberta.ca/imaging/snfs/.

[8] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4:187–217, 1983.

[9] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[10] B. Callaghan. *NFS Illustrated*. Addison-Wesley, 2000.

[11] R. Coker. The Bonnie++ benchmark. http://www.coker.com.au/bonnie++/.

[12] M. Ding and P. Lu. Trellis-SDP: A Simple Data-Parallel Programming Interface. In *3rd Workshop on Compile and Runtime Techniques for Parallel Computing (CRTPC) held with the 33rd International Conference on Parallel Processing (ICPP-04)*, pages 498–505, August 2004.

[13] R. Figueiredo, N. Kapadia, and J. Fortes. The PUNCH Virtual File System: Seamless access to decentralized storage services in a computational grid, 2001.

[14] A. S. Grimshaw, Wm. A. Wulf, and the Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[15] D. Hitz and A. Watson. The Evolution of NFS. Technical report, Network Appliance, Inc., 2002.

[16] R. Jones. Netperf. http://www.netperf.org/netperf/NetperfPage.html.

[17] M. Kan, D. Ngo, M. Lee, P. Lu, N. Bard, M. Closson, M. Ding, M. Goldenberg, N. Lamb, R. Senda, E. Sumbar, and Y. Wang. The Trellis Security Infrastructure: A Layered Approach to Overlay Metacomputers. In *The 18th International Symposium on High Performance Computing Systems and Applications*, 2004.

[18] E. Lindahl, B. Hess, and D. van der Spoel. GROMACS 3.0: A package for molecular simulation and trajectory analysis. *J. Mol. Mod.*, 7:306–317, 2001.

[19] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[20] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 85–105, Edinburgh, Scotland, UK, July 24 2002. Also published as Springer-Verlag LNCS 2537 (2003), pages 205–228. Available at http://www.cs.ualberta.ca/~paullu/.

[21] C. Pinchak, P. Lu, J. Schaeffer, and M. Goldenberg. The Canadian Internetworked Scientific Supercomputer. In *17th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 11–14 2003. Available at http://www.cs.ualberta.ca/~paullu/.

[22] R. Rivest. *RFC1321: The MD5 Message-Digest Algorithm*. Network Working Group of the IETF, April 1992.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[24] M. Shand, D. Becker, R. Sladkey, O. Zborowski, F. van Kempen, and O. Kirch. The Linux UNFSD Server. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir/.

[25] J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 480–483, Chicago, Illinois, USA, September 23–36 2002. Available at http://www.cs.ualberta.ca/~paullu/.

[26] H. Stern. *Managing NFS and NIS*. O'Reilly, 1991.

[27] Sun Microsystems and others. The Connectathon NFS Testsuite. http://www.connectathon.org.

[28] Sun Microsystems, Inc. *RFC1094: NFS: Network File System Protocol Specification*. Network Working Group of the IETF, March 1989.

[29] B. S. White, A. S. Grimshaw, and A. Nguyen-Tuong. Grid-based File Access: The Legion I/O Model. In *IEEE International Symposium on High-Performance Distributed Computing*, pages 165–174, 2000.

[30] T. Ylonen. SSH - Secure login connections over the internet. *Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA)*, 1996.