Error Management in the Pluggable File System

Douglas Thain
Miron Livny

Technical Report #1448

October 2002

Error Management in the Pluggable File System

Douglas Thain and Miron Livny

9 October 2002
Technical Report 1448
Computer Sciences Department
University of Wisconsin

Abstract

Distributed computing continues to be an alphabet-soup of services and protocols. No single system for managing CPUs or I/O devices has emerged (or is likely to emerge) as a universal solution. Therefore, distributed applications require adapters in order to plug themselves into existing systems. The difficulty of building such adapters lies not in normal operations, but in the complications of failures and other unusual situations. We demonstrate this with the Pluggable File System, an adapter for connecting POSIX applications to remote I/O services. We offer a detailed discussion of the construction of the system while dealing with failures and other events that are not trivially mapped into the application's expectations. The key insight is that correct I/O management requires coordination with CPU management. We conclude with some practical advice for others constructing similar software. [1]

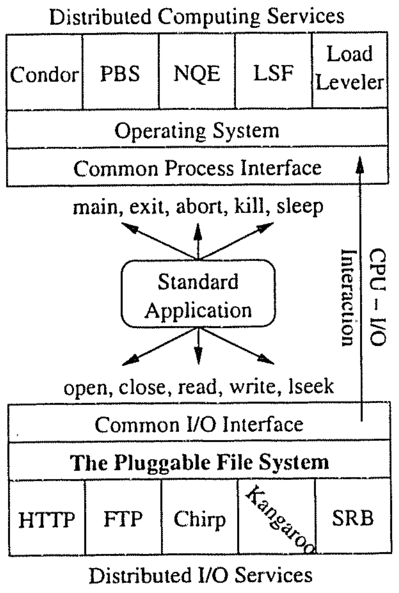

Figure 1. The Hourglass Model

1. Introduction

The field of distributed computing has produced countless systems for harnessing remote CPUs and accessing remote data. Despite the intentions of their designers, no single system has achieved universal acceptance or deployment. Each carries its own strengths and weakness in performance, manageability, and reliability. A renewed interest in world-wide computational systems called grids [17] is increasing the number of systems and interfaces in play. A complex ecology of distributed systems is here to stay.

The result is an *hourglass model* of distributed computing, shown in Figure 1. Users submit batch jobs to a variety of different interfaces. Each system interfaces with a process through standard POSIX interfaces such `main` and `exit`. This interface is so simple that it is rarely discussed and has no name, yet it is certainly universal and critical to the wide deployment of applications across batch systems. Equally important is the common interface to I/O services. An operating system transforms standard operations such as `read` and `write` into the low-level block and network operations needed for a local or distributed file systems.

Yet, a common interface to I/O operations is not enough. Applications require common conventions for naming and data access beyond the simple names of the I/O functions. The wide variety of systems available to a user through remote batch systems all have varying degrees of access to local and distributed systems scattered across the grid. Although many file systems aim to provide a universal naming scheme across the entire internet, none are actually deployed to this degree. Without universal naming and data access, a common interface has little value.

To remedy this situation, we advocate the use of *inter-*

[1]This research was supported in part by a Cisco Distinguished Graduate Fellowship and a Lawrence Landweber NCR Fellowship in Distributed Systems.

position agents, or sometimes *adapters* for short. These devices transform standard interfaces such as POSIX I/O into remote I/O protocols not normally found in an operating system kernel. In effect, an adapter allows an application to bring its filesystem along wherever it goes. This releases the dependence on the kernel details of the execution site while preserving the use of standard interfaces.

In this paper, we present the Pluggable File System (PFS), a user-level interposition agent that transforms an application's standard POSIX I/O operations into a variety of remote I/O services. We describe the overall architecture and naming scheme of the system, and offer some practical discussion on the fine details necessary to make PFS operate with real applications.

The multiplexing of a standard interface is a standard technique in any programmer's repertoire. In the realm of I/O, the mapping is quite simple: `read` becomes a `read`, `write` becomes a `write`, and so on. The real difficulty lies in the vast new kinds of failure in a distributed system: servers crash, networks fail, and disks fill. Furthermore, applications frequently require operations that have no obvious analogue in a remote system. How can such difficulties be reconciled with the "no-futz" requirements of large-scale distributed computing?

Our primary contribution is a detailed discussion of the problem of new error modes. We are not exploring techniques of *fault tolerance*, roughly defined as masking failures through retry or reservation of extra capacity. Rather, we are studying the problem of *error management*, in which we seek to simple direct an honest message about a failure along the correct channel. The key to error management, as suggested in Figure 1, is to recognize that correct I/O management must carefully interact with process management. The two do not stand alone.

To discuss these problems in detail, we must necessarily present the Pluggable File System in a fair amount of detail. We begin by describing the architecture and capabilities of PFS. We then move to examine how to handle the problems of missing operations and unusual failure modes. We conclude with a discussion of some practical problems that arose in the construction of PFS, in the hope that it will assist others building such systems.

2. User's Experience

To offer some of the flavor of PFS, we begin by outlinging how a user might interact with it in an interactive setting. The **pfsrun** program starts a new application with PFS attached:

```
% pfsrun tcsh
```

With PFS attached, the local filesystem is still visible through all of its normal filenames:

```
% cat /etc/passwd
% grep word /usr/dict/words
```

In addition, remote resources appeach under names in the root directory reflecting the protocol used to access them. These are quite similar to the Uniform Resource Locators used by the world wide web and other applications.

```
% vi /http/www.yahoo.com/index.html
% less /ftp/ftp.cs.wisc.edu/RoadMap
```

Finally, remote directory trees may be spliced into the local filesystem through the use of a *mount list*. Each entry in the list consists of a name in the logical file system to be redirected to a remote directory or file. Here is a simple mount list:

```
/in      /chirp/nest.wisc.edu/indir
/out     /kangaroo/kang.wisc.edu/outdir
```

And here's how it might be used:

```
% pfsrun -mountlist mount.file tcsh
% grep function /in/*.c > /out/results
```

3. Architecture

PFS is built with Bypass [35], a general-purpose tool for building interposition agents. It takes the form of a shared library which may be forcibly inserted into any dynamically linked process. This capability is present in most Unix-like operating systems, although it is activated in a variety of ways. `pfsrun` is a script which hides such details from the user. Bypass preserves all of the existing entry points to POSIX functions at the standard library interface, so PFS is free to use ordinary libraries and standard routines internally. It makes use of large, general-purpose library such as the Secure Sockets Layer and the Globus Toolkit [16] without modifications.

Figure 2 shows the internal structures of PFS. They are quite similar to the process I/O structures in a standard operating system kernel, so PFS might be considered a *virtual operating system*. Strictly speaking, PFS does not modify any files directly. A *file* is a persistent data structure on stable storage with attributes such as a size, owner, creation time, and so on. PFS has many internal data structures that represent files, but relies on other systems to actually examine and modify the files themselves.

PFS has four levels of data structures that represent files.

A *file descriptor* (fd) is an integer allocated by **open** or **dup** and is the only user-visible handle to an open file. A file descriptor contains no identifying information about a file, but only serves as a reference to a file pointer below.

A *file pointer* records the *current seek pointer* (csp) used to remember an application's position in a file. The csp is

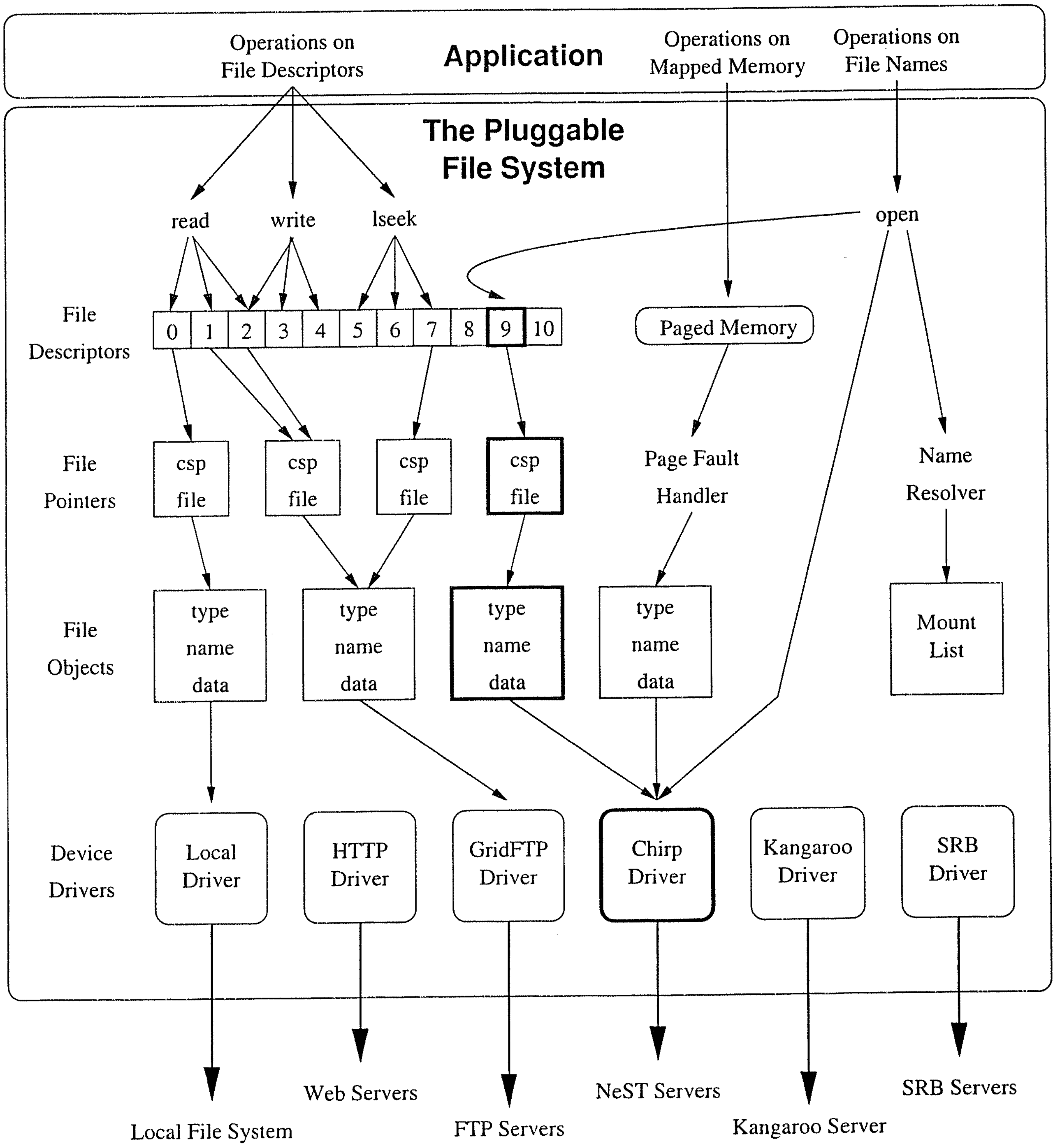

Figure 2. Architecture of the Pluggable File System

modified by operations such as **lseek** and is implicitly used by operations such as **read** and **write** to determine what portion of a file to examine or modify. A file pointer also refers to a file object below.

A *file object* represents a file currently in use by the application. It records the type of storage system where the file is stored, the name of the file, and any other private data, such as open sockets, necessary to access the file. It refers to a device driver below to actually perform file operations.

A *device driver* represents an entire system or protocol for accessing data, such as HTTP or simply the local file system. It implements all of the I/O operations that must be passed to a remote system. Notice that there is no generic data structure that represents an open network connection or a single remote server. Such details are encapsulated inside the device driver, as the number and type of remote connections necessary is quite dependent on the details of the protocol involved.

As Figure 2 suggests, data structures are shared at every level. For example, file descriptors 1 and 2 share a file pointer. This would occur if file descriptor 2 was created by a call to **dup**. Also, file descriptors 1,2, and 7 all share the same file object. This would occur by a call to **open** with the same file name used to access file descriptor 1.

PFS has a large number of entry points for all variety of I/O operations. However, we may classify the large majority of them into two categories: operations on file descriptors and operations on file names. The former traverse most of the data structures in PFS, while the latter take a more direct route to the device drivers.

Operations such as **read**, **write**, and **lseek** operate on file descriptors. Upon entering PFS, these commands check the validity of the given fd, and then descend the various data structures. **read** and **write** take the csp from the corresponding file pointer and use it as an argument to call a **read** method in the corresponding file object. The file object, through the device driver, performs the necessary remote operation.

Other operations such as **rename**, **stat** and **delete** operate on file names. Upon entering PFS, these commands first pass through the *name resolver*, which may transform the program-supplied name (or names) according to a variety of rules and systems. The name resolver is discussed in more detail below. Then, the transformed names are passed directly to the device driver, which performs the operation on the remote system.

open is a special case. Figure 2 shows how it interacts with the system in three ways. First, it transforms the given name using the name resolver and the mount list, if any. Second, it contacts the named device driver and attempts to open the file. Third, if successful, it allocates a file descriptor, pointer, and object according to the newly open file and installs them in the tree of data structures. The bold items in Figure 2 highlight a newly opened file at descriptor 9 using the Chirp driver.

Most UNIX applications access file through explicit operations such as **read** and **write**. However, files may also be memory mapped. In a standard operating system, a memory mapped file is a separate virtual memory segment whose backing store is kept in the file system rather than in the virtual memory pool. PFS accomplishes the same thing using its own underlying drivers, thus reducing memory mapped files into the same mechanisms as other open files.

When the user establishes a memory mapping with the **mmap**, PFS allocates a corresponding piece of virtual memory from its own heap via the standard **malloc** allocator. However, the application's permission to read and write the memory are removed by using **mprotect**. When the user attempts to access the memory, the system raises a **SIGSEGV** signal indicating a memory access violation. PFS traps this signal and uses the **SA_INFO** option to extract the address of the memory reference. If the address falls within a memory mapped file managed by PFS, it passes control to the page fault handler shown in Figure 2.

The page fault handler currently performs demand paging with writeback at close. As read page faults occur, they are satisfied by issuing **read** operations on the underlying driver. As write faults occur, data are simply written the local memory region. When the user calls **munmap** to delete the segment, dirty pages are written back to the target storage. Naturally, the default page size used by the underlying system is much too small for the latency of I/O operations over the wide area network. The user may select an appropriate page size through an environment variable.

This memory-mapping facility is quite simple. It does not perform any pageouts in order to fit the segment into physical memory. Rather, the whole segment is stored in virtual memory, under the assumption that paging out to local disk is much faster than paging out to remote storage. In addition, there is no facility for enforcing coherence on processes the communicate via a shared memory segment. This is generally impossible, as most remote I/O protocols have no facilities for coherence. However, this has not proven to be a significant obstacle, as our primary target of sequential scientific applications (by definition) do not require distributed shared memory.

4. Drivers

PFS is equipped with a variety of drivers for communicating with external storage systems. The C++ interface to a driver is shown in Figure 3. This interface lets the user perform single operations on named files. Two methods in the interface bear explanation: **open** and **getdir**.

The **open** method found in the driver is a factory method that binds a file name to a file object, which is shown in

```
class pfs_driver {

  pfs_file * open( path, flags,  mode );
  pfs_dir  * getdir( path );

  int        stat( path, buf );
  int        lstat( path, buf );
  int        unlink( path );
  int        access( path, mode );
  int        chmod( path, mode );
  int        rename( oldname, newname );
  int        chdir( path );
  int        readlink( path, buf, bufsiz );
  int        mkdir( path, mode );
  int        rmdir( path );

};
```

Figure 3. Driver Interface

```
class pfs_file {

  int          close();
  pfs_ssize_t  read( data, length, pos );
  pfs_ssize_t  write( data, length, pos );
  int          stat( buf );
  int          truncate( length );
  int          sync();
  int          fcntl( cmd, arg );
  int          ioctl( cmd, arg );
  int          fchmod( mode );

  pfs_ssize_t  get_size();
  int          is_seekable();
  int          is_local();
  int          isatty();

};
```

Figure 4. File Interface

```
class pfs_dir {

  struct dirent * read();
  void            rewind();
  pfs_size_t      tell();
  void            seek( pfs_size_t pos );
  int             append( char *name );

}
```

Figure 5. Directory Interface

Figure 4. Once opened, the file object serves as the focal point for operations on file descriptors or mapped memory. To support access through multiple file pointers at once, the file object appears to be random access: it accepts an offset argument to **read** and **write**. However, not all driver types actually support random access to files. For this reason, an additional method, **is_seekable** is used by PFS to determine whether an **lseek** on such a file will succeed. The consequences of attempting to **lseek** within a sequential-only driver are explored below.

The **getdir** method found in the driver does not correspond to any single function normally found in the standard library or at the kernel interface. Few remote I/O systems have a stateful set of commands for scanning a directory, such as **opendir**, **readdir**, and **closedir**. Instead, a single atomic operation retrieves a whole directory listing. The upper layers of PFS are resonsible for implementing the stateful POSIX directory-scanning commands.

Each of the various drivers in PFS has some special processing and unusual cases. Let's explore each of them in turn.

4.1. Local

The local driver is very simple. It passes all file system operations directly through to the underlying kernel. The local driver is used by default when a file name does not map to any remote system. Thus, a program using PFS still has access to all data in the local filesystem. This comes with very little overhead. The trapping mechanism provide by Bypass only adds a few microseconds to every call. [35] By default, memory mapped to a local file uses the native memory-mapping mechanism for performance. However, the user-level mechanism provided by PFS may optionally be enabled in order to trace an application's memory access patterns.

4.2. HTTP

The HTTP driver is also quite simple, but provides much less functionality than the local driver. HTTP simply doesn't support many of the operations necessary for a general file system. Arbitrary files may be read sequentially. The **stat** command may be used to examine a file, but the only available information is the size of the file. Directory listings are not possible. HTTP performs whole file transfers, therefore every concurrent open file requires its own connection. After a file is closed, its connection is cached by the driver for possible future use.

Other interposition systems such as UFO [4] have used whole-file fetching of files available through sequential-access protocols such as HTTP, thus simplifying the problems of random access and permitting a cache of recentlyused files. We have not taken this route for two reasons. First, whole-file fetching introduces a large latency when a file is first opened. This is an unnecessary price when an application could take advantage of overlapped CPU and I/O access by reading streamed files sequentially. Second, few remote I/O protocols have a reliable mechanism for ensuring synchronization between shared and cached files. The user who is willing to deal with both of these problems may explicitly make a local copy (via PFS, of course) and then operate on it directly.

4.3. FTP

In contrast to HTTP, the FTP driver is very full-featured. Depending on the variant of the server, sequential access, directory listings, and querying of meta-data may be possible. If a server requires a password, an interactive user may type it at the console while the application is blocked. These features come at considerable complexity. Many file operations require attempting several different command variations in order to support the many flavors of the protocol. In addition, each open file requires its own interaction with the server: one TCP stream for control, and one TCP stream for data.

The FTP driver also supports the GSI-enabled FTP [5] variant. This protocol introduces strong authentication to remote services without using cleartext passwords. In addition, more efficient commands for partial-file access are available, along with high-throughput sequential access via multiple TCP streams.

4.4. Chirp

The Chirp protocol, spoken by the NeST storage appliance, is somewhat easier to integrate with PFS due to its similarity with the POSIX interface. Chirp permits random access to arbitrary files, meta-data requests, directory listings, and more. In addition, access to multiple files may be interleaved on the same connection, so the driver only needs to maintain one connection per server accessed, rather than one per open file.

The Ghost driver is a research variant of the Chirp driver. This driver redirects operations to a nearby ghost, which is a localized buffer cache for a remote NeST storage appliance. A set of ghosts with a parent NeST forms a *migratory file service*, which is discussed further by Bent, et al. [7]

4.5. Kangaroo

The Kangaroo protocol is even easier to integrate, although it offers less power than Chirp. The Kangaroo system is designed to offload all of an application's I/O requests to a single nearby server, where they can be satisfied through buffering, caching, and remote I/O. Thus, the Kangaroo device driver simply performs trivial RPCs on a single server. No connection management is necessary. Kangaroo does not support directory listings.

A unique feature of Kangaroo is its data-consistency protocol. Unlike many remote file services, Kangaroo provides no confirmation of write operations until the client issues an explicit **commit** or **push** command. The former forces data to stable storage at the nearest server, while the latter blocks until it is visible at its destination. PFS issues a **commit** when the application exits and optionally performs **push** in response to an **fsync**.

5. Error Handling

Error handling has not been a pervasive problem in the design of traditional operating systems. As new models of file interaction have developed, their attending error modes have been added to existing systems by expanding the software interface at every level. For example, the addition of NFS [32] to the Unix kernel created the new possibility of a stale file handle, represented by the **ESTALE** error. As this error mode was discovered at the very lowest layers of the kernel, the value was added to the device driver interface, the file system interface, the standard library, and expected to be handled directly by applications.

We have no such luxury in PFS. Applications use the existing POSIX interface, and we have no desire or facility for changing it. Yet, the underlying device drivers generate errors ranging from the vague "file system error" to the bizarre "server's credentials have expired." How should the unlimited space of errors in the lower layers be transformed into the fixed space of errors available to POSIX?

Before we answer this question, we must remind the reader of our application domain. PFS was motivated by the need for scientific applications to access a variety of storage devices from a high-throughput batch execution system. In such a context, there are already many ways for a job to fail without the help of the I/O system: the submitter may lost contact with the execution site; the execution site may crash; the job may be forcibly evicted by the machine owner; and so on. Regardless of what may happen to the job, it is the responsibility of another process to oversee its progress and restart it if it should fail. Therefore, it is no disaster to kill the job when no other course seems reasonable.

This is not to say that killing the process is always the best solution. Rather, we must perform triage – some injured processes are not worth the trouble to save. We may divide errors into three general categories:

1. A *transformable error* may easily be converted into a form that is both honest and recognizable by the application. Such errors are converted into an appropriate

`errno` and passed up to the application in the normal way. Some transformable errors take considerable effort to determine the precise reason for the error.

2. A *permanent error* indicates that the process has a fatal flaw and cannot possibly run to completion. With this type of error, PFS must halt the process in a way that makes it clear the CPU system must not reschedule it.

3. A *transient error* indicates the process cannot run here and now, but has no inherent flaw. When encountering transient errors the I/O system must interact with the CPU system. It must indicate that the job is to release the CPU, but would like to execute again later and retry the I/O operation.

The handling of errors requires interaction between CPU and I/O managers. In order to handle both permanent and transient errors correctly, the I/O system must inform the CPU system exactly what the next course of action for the process must be. In the case of the permanent error, the I/O system must forcibly halt the process in a manner that cannot be misinterpreted by the CPU manager. In most batch systems, this is accomplished by terminating normally with a non-zero exit code. In the case of a transient error, the situation is more complex. We would like to attach complex conditions to the restart of a process. For example, restart could be triggered by the arrival of a file or the completion of another process. However, we may minimally satisfy our need with a simple hook for yielding the CPU and allowing another process to be scheduled in the batch system. In Condor, this is accomplished by terminating abnormally with a signal indicating outside interference. (i.e. SIGKILL) The batch system will retract the process and reschedule it at some future time on another CPU.

We must emphasize the difference between local CPU management and batch CPU management. In response to a transient error, PFS could simply block or sleep until the necessary data are available. This would indeed cause the running process to release the CPU and move to a wait state in the local operating system scheduler. However, what the process is actually doing with the CPU is irrelevant to the distributed batch system. Unless the program issues some explicit instruction to the batch system, it still is in posession of the CPU. It will continued to be charged (either in money or priority) for consuming the resource, whether it is actually consuming physical cycles or not.

Each of the three types of errors come from two distinct sources of errors. A mismatch of requests occurs when the target system does not have the needed capability. A mismatch of results occurs when the target system is capable, but the result has no obvious meaning to the application. Let's consider each in turn.

5.1. Mismatched Requests

Our first difficulty comes when a device driver provides no support whatsoever for an operation requested by the application. We have three different solutions to this problem, based on our expectation of the application's ability to handle an error. Representative examples are **unlink**, **lseek**, and **stat**.

Read-only services such as HTTP do not allow files to be deleted. A call to **unlink** a file cannot possibly succeed. Such a failure may be trivially represented to the calling application as "permission denied" or "read-only filesystem" without undue confusion by the user. Applications understand that **unlink** may fail for any number of other reasons on a normal filesystem, and are thus prepared to understand and deal with such errors.

In contrast, almost no applications are prepared for **lseek** to fail. It is generally understood that any file accessed through **open** may be accessed randomly, so few (if any) applications even bother to consider the return value of **lseek**. If we use **lseek** on an FTP server that does not implement random access through the REST command, we risk any number of dangers by allowing a never-checked command to fail. Therefore, an attempt to seek on a non-seekable file results in a permanent error, returning the job to the user with an message on the standard error stream.

The **stat** command offers the most puzzling difficulty of all. **stat** simply provides a set of meta-data about a file, such as the owner, access permissions, size, and last modification time. The problem is that few remote storage systems provide all, or even most, of this data. For example, FTP provides a file's size, but no other meta-data in a standard way. [2]

We initially caused **stat** to report "permission denied" on such systems, indicating the data were not available. But to our surprise, this caused a large majority of programs to fail. **stat** is a very frequent operation used by command-line tools, large applications, and even the standard I/O library. We were quite dismayed at this discovery, because it seemed the necessary information simply could not be extracted from most remote I/O systems. However, a brief investigation into the actual uses of **stat** gave some cause for hope. Here are some of its major applications:

- **Cataloging.** Commands such as **ls** and program elements such as file dialogs use **stat** to annotate lists of files with all possible detail for the interactive user's edification.

[2]A de-facto standard in FTP is the LIST -l command, which usually provides a detailed UNIX file list, actually performing a **stat** on every file in a directory. However, this cannot be relied upon, as each server provides a slightly different selection of attributes in a slightly different format. Further, not all servers are UNIX-like, and even those that are have no obligation to produce such output.

Device Number	=	0
Index Number	=	Incremented at every call
Permissions	=	RWX by anyone
Number of Links	=	1
User	=	The calling user
Group	=	The group of the calling user
Size	=	0
Block Size	=	User-configurable
Blocks	=	0
Last Access Time	=	Current time
Last Mod. Time	=	Current time
Last Change Time	=	Current time

Figure 6. Default Results of Stat

- **Optimization.** The standard C library, along with many other tools, uses **stat** to retrieve the optimal block size to be used with an I/O device. This is used to choose the buffer size for the ANSI buffered I/O interface.

- **Short-circuiting.** Many programs and libraries, including the command-line shell and the Fortran standard library, use **stat** or **access** as a quick way to check the presence or validity of a file before actually performing an expensive **open** or **exec**.

- **Unique identity.** Tools such as **cp**, which copy one file to another, use the unique device and file numbers returned by **stat** to determine if two file names refer to the same physical file.

In each of these cases, there is very little harm in presenting default, or even guessed information. No program can rely on the values returned by **stat** because it cannot be done atomically with any other operation. If a program uses **stat** to measure the existence or size of a file, it must still be prepared for **open** or **read** to return conflicting information. Therefore, we may fill the response to **stat** with benevolent lies that encourage the program to continue, whether for reading or writing. Each device driver fills in whatever values in the structure it is able to determine, and then fills the rest with defaults shown in Figure 6. Or course, if the device driver can inexpensively determine that the file actually does not exist (i.e. the FTP **SIZE** command or the Chirp **Status** command) then it may truthfully cause **stat** to fail.

The block size field shown in Figure 6 deserves special mention. In practice, the actual block size of the underlying device is irrelevant to the file abstraction. As we mentioned, it is instead used as an optimization parameter. The optimal block size for a remote protocol may be more of a property of the network and local environs than the remote storage

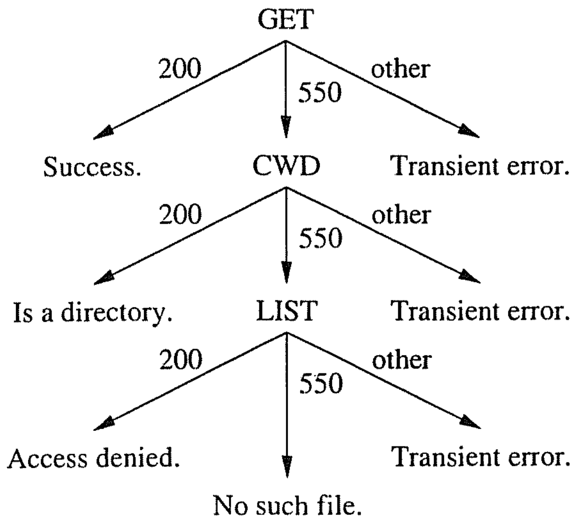

Figure 7. An Error Interview

device itself. So, PFS allows a file's blocksize to be chosen by the user through an environment variable, allowing the buffered I/O interface to seamlessly adapt to protocols requiring a large I/O granularity.

5.2. Mismatched Results

Several device drivers have the necessary machinery to carry out all of a user's possible requests, but provide vague errors when a supported operation fails. For example, the FTP driver allows an application to read a file via the GET command. However, if the GET command fails, the only available information is the error code 550, which encompasses almost any sort of file system error including "no such file," "access denied," and "is a directory." The POSIX interface does not permit a catch-all error value – it requires a specific reason. Which error code should be returned to the application?

One technique for dealing with this problem is to interview the service in order to narrow down the cause of the error. This is similar to a standard expert system or the functional error-interview system described in [13]. Figure 7 shows the interview tree for a GET operation. If the GET should fail, we assume the named file is actually a directory and attempt to change to it. If that succeeds, the error is "not a file." Otherwise, we attempt to LIST the named file. If that succeeds, the file is present but inaccessible, so the error is "access denied." If it fails, the error is finally "no such file."

The error interview technique also has some drawbacks. It significantly increase the latency of failed operations, although it is generally not necessary to optimize error cases. In addition, the technique is not atomic, so it may determine an incorrect value of the remote filesystem is simultaneously modified by another process.

There is also very large space of infrequent errors that simply have no expression at all in the application's interface. A NeST might inform PFS via Chirp that its disk allocation has expired and been deleted. PFS might discover that the connection to a Kangaroo server has been broken by a network failure. An FTP server may inform PFS that the backing storage is offline. User credentials, such as Kerberos or GSI certificates, may expire, and no longer be valid.

Of course, there are many well-known techniques for hiding such errors. Lots may be re-allocated, lost connections may be rebuilt, storage may come online again, and certificates might be renewed by the user. However, all of these techniques take time and computing resources and have no guarantee of eventual success. At some point, we must accept that an error has occured.

There is no honest way to report such errors to the application. Reporting "no such file" or "access denied" does not give the application the information necessary to recover when the true problem lies elsewhere. This represents a transient error that must be handled or re-tried by a higher layer of software. In these cases, PFS forces the process to exit abnormally.

6. Other Complications

A number of other situations arose in the development of PFS which deserve elaboration for builders of similar systems. These include the sharing of state between processes, the initialization of complex programs, and signal propagation in interposition agents.

6.1. Process Creation

So far, we have concentrated on the problem of serving a single process. PFS also works across the creation of new proceses. This is most useful in the context of an interactive shell, which may create connections to remote devices, and pass them implicitly as the standard input and output streams of a new process.

In a standard operating system, this is quite simple. Because all I/O structures are in the kernel, they are trivially shared between all processes. Things are more complicated in PFS, where each process must have its own I/O structures, yet still be able to share remote files. We cannot rely on the simple file descriptor inheritance provided by the operating system, because it preserves the wrong level of detail. For example, a device driver may hold a socket open to an FTP server. A child process cannot simply share this socket without harming the parent's interaction with the server. Instead, it must create a new connection by re-opening at the highest layers of PFS.

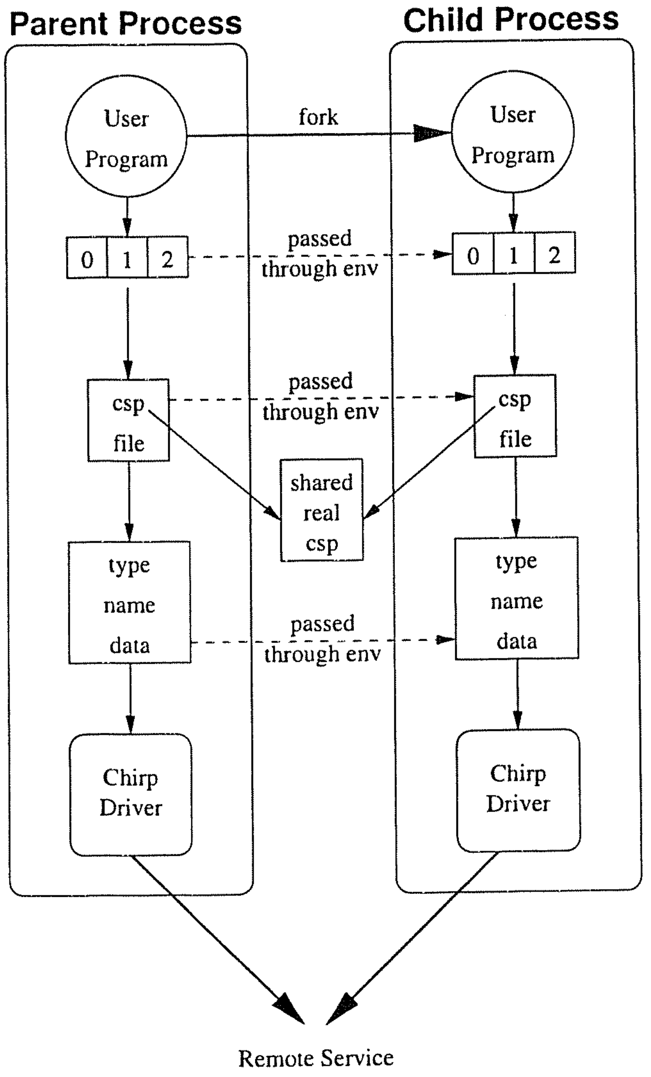

Figure 8. Sharing a File Between Processes

To accomplish this, PFS serializes all of its state when creating a new process. The child process is given an environment variable describing the state of all data structures, ranging from file descriptors down to device drivers. As the instance of PFS in the child process initializes, it re-creates the state of the parent by building all of the necessary data structures and re-opening files via the device drivers.

This technique allows the child process to begin with the same file state as the parent had when the child was created. However, as both processes run, they must continue to share some state in order that they may interact.

The simplest and most vital sharing is that of file pointers. A parent frequently creates a child with a shared output stream. If the child produces some output, thus advancing the current seek pointer, the parent must see the changes in order that its output may append to the child's, rather than overwriting it.

We may rely on the host OS for a solution to this problem. Before creating a new process, PFS allocates a shared csp from the OS by creating a dummy temporary file and then deleting it while still open. This file desciptor may be used for recording the csp of a file externally in a way visible to all processes that share it. It is also automatically de-allocated when the last process exits.

This shared csp is distinct from any underlying file descriptors necessary to perform data access. For example, a file accessed by HTTP and shared between two processes has one file descriptor in use as a shared csp and another file descriptor in use as the network connection to the HTTP server. The former is shared while the latter is private.

The actual sharing of data between files occurs at the remote service itself. Two related processes writing to a file will synchronize their position with a shared csp, but the actual combination of their write operations occurs at the remote service.

6.2. Program Initialization

Program initialization is a very complicated matter. Many programs rely on a portion of their code to be executed before the program's formal entry point (`main`) or after the program exits. These hooks include C++ constructors and destructors, the ANSI `atexit` system, and shared library initializers and finalizers.

Originally, PFS relied on C++ constructors and destructors to create and clean up all of the data structures necessary to support an application's I/O. This created a very puzzling intermittent problem. Depending on the operating system, compiler, and the time of day, a process coupled with PFS would mysteriously crash before it reached its entry point or after it exited. The problem was in the ordering of global constructors. Specifically, there wasn't one. A conforming C++ system call all of the global constructors in different translation units in any order it likes. Thus, there was no guarantee that PFS would initialize its state before the application. If an application's global constructor performed I/O and happened to be executed before PFS's constructor, a crash would occur.

The solution was to simply make PFS self-initializing. A static variable records whether the necessary data structures have been created. At the first call to any sort of I/O operation, the state is checked and the system is initialized. Thus, constructors could be called in any order with respect to PFS.

We submit that the general technique of application code executed implicitly by the system linker or loader is a bad idea. In fact, the otherwise dense ANSI C++ standard devotes several pages to this troublesome problem. (See Section 3.4 in [14]) Nearly any sort of complex initialization code has a dependency on another subsystem that *also* needs initialization. For example, the standard I/O library certainly depends on the standard memory allocator. To make computer systems work reliably, an ordering on initialization must be imposed.

This could be accomplished by the system linker (or loader.) At build time, the programmer could give a list of what systems are to be initialized in what order. However, engineering a large piece of software is complex enough already. We are very loath to suggest adding a dependency between the program code and the options given to the system linker. A better solution is to make all systems self-initializing. For example, all entry points to the standard I/O library may check to see if the standard streams have been initialized. There are well-known solutions to make such code thread-safe. If self-initialization causes an unnecessary overhead, then the subsystem may be initialized by an explicit call at the beginning of the program.

One might argue that such explicit initialization is an unnecessary burden on the programmer, and implicit initialization simplifies the program. We disagree. We have already shown that an explicit initialization list is necessary for program correctness. It is better to place it in the program itself as a portable system-independent list of initializers than to place it in the linker or loader, where it is sure to be non-portable.

6.3. Signal Handling

In the concluding remarks of our paper describing Bypass, [35], we note that signal handling is not yet integrated with the interpositioning technique. Bypass uses a *current layer pointer* to track where the program is in the stack composed by the application, interposition agents, and standard library. The arrival of a signal (incorrectly) does not affect the current layer pointer. Therefore, a signal-handling function may call code in an incorrect layer.

If this is not clear, consider the same problem in another context. An operating system relies on the underlying hardware to switch to supervisor mode when a device interrupt is raised. Bypass does not switch to supervisor mode (i.e. the PFS layer) when a software signal is raised.

PFS relies on the correct operation of signals to implement memory-mapped files. To work around this limitation, PFS uses two hooks exposed by Bypass to manipulate the current layer pointer directly. When establishing a memory mapped file, PFS saves the layer pointer corresponding to itself. Upon entering the page fault handler, PFS saves the current layer pointer, and temporarily makes itself the current layer. It may then service the page fault using all of its machinery and the standard library below. Finally, the current layer pointer is restored before exiting the handler.

7. Related Work

Distributed batch systems are perhaps one of the oldest general-purpose applications of distributed computing. Many, such as the Cambridge Ring [27], evolved as a response to the expense of centralized computing systems. Today, a large number of such systems are deployed at commercial and academic sites, including Condor [25], LoadLeveler [1] (a descendant of Condor), LSF [38], Maui [22], NQE [3], and PBS [20]. Several application-specific batch systems have been built to solve specific problems, such as SETI@Home [2] and the RC5 [23] challenges.

Distributed I/O has traditionally been the realm of file systems. However, we must emphasize that the formula of a distributed file system as kernel-provided resource does not match the environment of mistrust and minimalism present in most distributed batch systems. When borrowing CPU time from a remote (and possibly anonymous) machine owner, it is simply not possible to request changes in the kernel. I/O systems accessible to user-space processes have generally fallen into two categories. Protocols and systems such as HTTP [15], FTP [28], GridFTP [5], and GASS [9] have cast themselves as protocols and interfaces for high-throughput whole-file movement. Other systems and protocols such as Condor remote I/O [25], RIO [18], Chirp [8], and Kangaroo [33] perform fine-grained access and remote files without extensive caching or transfer overhead. Both models work well with PFS.

Of course, systems other than POSIX may sit at the center of the hourglass. Java, MPI, and PVM all have significant user communities and have found support in various batch systems such as Condor [36, 37, 29]. Although this paper relies on POSIX for concrete examples, much of it applies to other execution systems. We have already discussed some I/O problems unique to Java in [36].

The term *interposition agent* was coined by Michael Jones. A number of techniques for a building interposition agents have been devised, and an excellent review is given by Alexandrov, et al. in conjunction with the UFO [4] system. Other general-purpose toolkits include Detours [21], mediating connectors [6], and SLIC. [19] Component systems such as Knit [31] solve the problem of inter-component initialization by making component dependences explicit to the linker.

Multiplexing of I/O devices is a common technique and is seen in devices as diverse as the in-kernel Virtual Filesystem Switch (VFS) [24], the user-level Uniform I/O Interface (UIO) [10], and the server-side NeST [8] multi-protocol layer. Multiplexing is found in many other contexts such as the GRAM [12] interface to batch execution, the Proteus [11] multiprotocol message library, and the Ace [30] system language for customizable shared-memory algorithms.

Despite the ubiquity of this technique, we are not aware of any detailed treatment regarding failures and interface mismatches when forced to used existing interfaces. The closest such discussion is a report by Craig Metz [26] on the correct use of the multiplexed Berkeley sockets interface.

8. Conclusion

The Pluggable File System has been used in a variety of research settings in the Condor project, including research into I/O communities [34], migratory file systems [7], and distributed buffering. [33] It continues to be a key component of our toolkit for research in distributed systems. We plan to gradually expand PFS to production settings and add support for more storage drivers.

Our contribution is an illumination of a unique aspect of software engineering: error management. Although interpositioning and multiplexing are standard techniques, the emphasis is usually placed upon the transformation of requests and not the interpretation of the results. The results of file system operations contain important information and cannot be casually discarded.

We have emphasized the importance of the narrow communication channel between I/O systems and CPU systems. Although both are designed in isolation, they require a certain level of integration in order to operate correctly. PFS has a relatively simple interaction with CPU managers through such operations as `exit` and `kill`. If designed carefully, a richer interface would allow for powerful interactions while preserving the design independence of I/O and CPU systems.

Manuals, software, and more details about the Pluggable File System may be found at `http://www.cs.wisc.edu/^condor/pfs`.

References

[1] *IBM Load Leveler: User's Guide*. I.B.M. Corporation, September 1993.

[2] Astronomical and biochemical origins and the search for life in the universe. In *Proceedings of the 5th International Conference on Bioastronomy*. Editrice Compositori, Bologna, Italy, 1997.

[3] Introducing NQE. Technical Report 2153_2.97, Cray Inc., Seattle, WA, February 1997.

[4] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[5] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, pages 161–163, 2000.

[6] R. Balzer and N. Goldman. Mediating connectors. In *19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[7] J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Migratory file services for scientific applications. In preparation, October 2002.

[8] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. A. Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[9] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. *6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[10] D. Cheriton. UIO: A uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February.

[11] K. Chiu, M. Govindaraju, and D. Gannon. The Proteus multiprotocol library. In *Proceedings of the Conference on Supercomputing*, November 2002.

[12] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. Resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[13] K. Efe. A proposed solution to the problem of levels in error-message generation. *ACM Computing Practices*, 30(11):948–955, November 1987.

[14] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1992.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol (HTTP). Internet Engineering Task Force Request for Comments (RFC) 2616, June 1999.

[16] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[17] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[18] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 14–25, 1997.

[19] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, June 1998.

[20] R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[21] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, February 1999.

[22] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, 2001.

[23] B. Kaliski and Y. L. Yin. On the security of the RC5 encryption algorithm. Technical Report TR-602, RSA Laboratories, September 1998.

[24] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX Technical Conference*, pages 151–163, 1986.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[26] C. Metz. Protocol independence using the sockets API. In *Procedings of the USENIX Technical Conference*, June 2002.

[27] R. Needham and A. Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley, London, 1982.

[28] J. Postel. FTP: File transfer protocol specification. Internet Engineering Task Force Request for Comments (RFC) 765, June 1980.

[29] J. Pruyne and M. Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[30] M. Raghavachari and A. Rogers. Ace: a language for parallel programming with customizable protocols. *ACM Transactions on Computer Systems (TOCS)*, 17(3), August 1999.

[31] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, San Diego, California, October 2000.

[32] 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 Conference*, pages 119–130, 1985.

[33] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California, August 2001.

[34] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Gathering at the well: Creating communities for grid I/O. In *Proceedings of Supercomputing 2001*, Denver, Colorado, November 2001.

[35] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[36] D. Thain and M. Livny. Error scope on a computational grid. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2002.

[37] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunisitc and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution*, Champaign-Urbana, Illinois, June 2001.

[38] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, December 1993.