This paper has appeared in a number of forms. It was originally peer-reviewed and published in limited quantity as:

- Douglas Thain and Miron Livny, "Parrot: Transparent User-Level Middleware for Data-Intensive Computing," *Workshop on Adaptive Grid Middleware* New Orleans, LA, September 2003.

It was later released as an identical technical report:

- Douglas Thain and Miron Livny, "Parrot: Transparent User-Level Middleware for Data-Intensive Computing," Technical Report 1493(a), Computer Sciences Department, University of Wisconsin, December 2003.

A condensed version will appear as:

- Douglas Thain and Miron Livny, "Parrot: An Application Environment for Data-Intensive Computing," *Journal of Parallel and Distributed Computing Practices,* to appear in 2004.

And, an extended version including application experience will appear as:

- Douglas Thain, Sander Klous, and Miron Livny, "Deploying Complex Applications in Unfriendly Distributed Systems with Parrot,"*Kluwer Journal of Supercomputing,* to appear in 2004.

Parrot: Transparent User-Level Middleware for Data-Intensive Computing

Douglas Thain and Miron Livny
Computer Sciences Department, University of Wisconsin

Abstract

Distributed computing continues to be an alphabet-soup of services and protocols for managing computation and storage. To live in this environment, applications require middleware that can transparently adapt standard interfaces to new distributed systems; such software is known as an interposition agent. In this paper, we present several lessons learned about interposition agents via a progressive study of design possibilities. Although performance is an important concern, we pay special attention to less tangible issues such as portability, reliability, and compatibility. We begin with a comparison of seven methods of interposition, focusing on one method, the debugger trap, that requires special techniques to achieve acceptable performance on popular operating systems. Using this method, we implement a complete interposition agent, Parrot, that splices existing remote I/O systems into the namespace of standard applications. The primary design problem of Parrot is the mapping of fixed application semantics into the semantics of the available I/O systems. We offer a detailed discussion of how errors and other unexpected conditions must be carefully managed in order to keep this mapping intact. We conclude with a evaluation of the performance of the I/O protocols employed by Parrot, and use an Andrew-like benchmark to demonstrate that semantic differences have consequences in performance. [1]

1. Introduction

The field of distributed computing has produced countless systems for harnessing remote processors 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. Renewed interest in world-wide computational systems is increasing the number of protocols and interfaces in play. A complex ecology of distributed systems is here to stay.

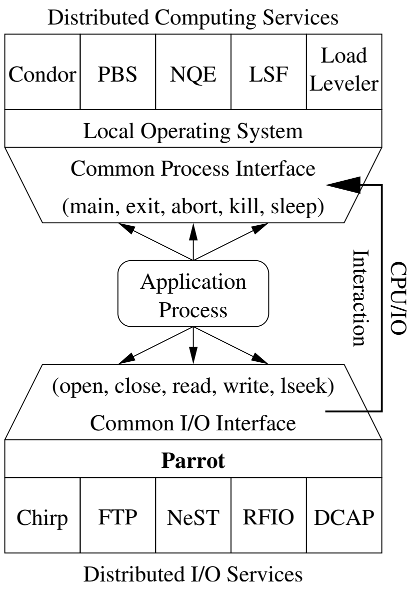


Figure 1. The Hourglass Model

The result is an hourglass model of distributed computing, shown in Figure 1. At the center lie ordinary applications built to standard interfaces such as POSIX. Above lie a number of batch systems that manage processors, interact with users, and deal with failures of execution. A batch system interacts with an application through simple interfaces such as *main* and *exit*. Below lie a number of I/O services that organize and communicate with remote memory, disks, and tapes. An ordinary operating system (OS) transforms an application's explicit *reads* and *writes* into the low-level block and network operations that compose a local or distributed file system.

However, attaching a new I/O service to a traditional OS is not a trivial task. Although the principle of an extensible OS has received much attention in the research community [26], production operating systems have limited facilities for extension, usually requiring kernel modifications or administrator privileges. Although this may be acceptable for a personal computer, this requirement makes it difficult or impos-

[1] This research was supported in part by a Lawrence Landweber NCR fellowship in distributed systems.

sible to provide custom I/O and naming services for applications visiting a borrowed computing environment such as a timeshared mainframe, a commodity computing cluster, or an opportunistic workgroup.

To remedy this situation, we advocate the use of *interposition agents* [17]. These devices transform standard interfaces into remote I/O protocols not normally found in an operating system. In effect, an agent allows an application to bring its filesystem and namespace along with it wherever it goes. This releases the dependence on the details of the execution site while preserving the use of standard interfaces. In addition, the agent can tap into naming services that transform private names into fully-qualified names relevant in the larger system.

In this paper, we present practical lessons learned from several years of building and deploying interposition agents within the Condor project. [27, 38, 31, 29, 32] Although the notion of such agents is not unique to Condor [17, 2, 16], they have seen relatively little use in other production systems. This is due to a variety of technical and semantic difficulties that arise in connecting real systems together.

We present this paper as a progressive design study that explores these problems and explains our solutions. We begin with a detailed study of seven methods of interposition, five of which we have experience building and deploying. The remaining two are effective but impractical because of the privilege required. We will compare the performance and functionality of these methods, giving particular attention to intangibles such as portability and reliability. In particular, we will concentrate on one method that has not been explored in detail: the debugger trap. Although this method has been employed in idealized operating systems, it requires additional techniques in order to provide acceptable performance on popular operating systems with limited debugging capabilities, such as Linux.

Using the debugger trap, we focus on the design of Parrot, an interposition agent that splices remote I/O systems into the filesystem space of ordinary applications. A central problem in the design of an I/O agent is the semantic problem of mapping not-quite-identical interfaces to each other. The outgoing mapping is usually quite simple: *read* becomes a *get*, *write* becomes a *put*, and so forth. The real difficulty lies in interpreting the large space of return values from remote services. Many new kinds of failure are introduced: servers crash, credentials expire, and disks fill. Trivial transformations into the application's standard interface lead to a brittle and frustrating experience for the user.

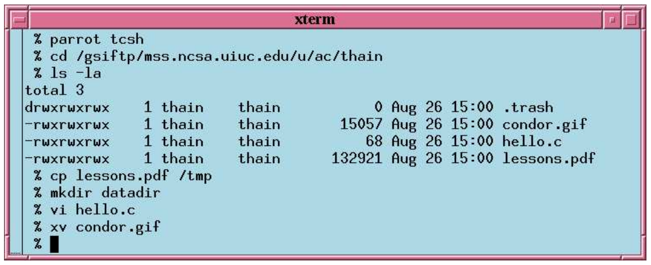
A corollary to this observation is that access to computation and storage cannot be fully divorced. Abstract notions of design often encourage the partition of distributed systems into two activities: either computation or storage. An interposition agent serves as a connection between these two concerns; like an operating system kernel, it manages both types of devices and must mediate their interaction, sometimes bypassing the application itself.

We deliberately postpone evaluating performance until after considering errors and other boundary conditions. This is because the correct handling of such cases can have significant performance implications. We will see how the detailed error interface of POSIX can result in a significant latency as a remote system is probed multiple times to extract the necessary information. We demonstrate the performance of Parrot with an Andrew-like benchmark and illustrate the importance of low latency operations for real applications.

2. Applications of Interposition Agents

Before diving into the many details of building interposition agents, we wish to give some examples of how they may be used in a distributed system.

Interactive browsing. The construction of a new type of storage protocol or device is frequently accompanied by the construction of tools to visualize, organize, and manipulate the contents. This is both time consuming and wasteful, as many tools already exist for these tasks on standard filesystems. As shown in Figure 2, Parrot enables ordinary tools to browse a remote archive, in this case the hierarchical mass storage server (MSS) at the National Center for Supercomputing Applications (NCSA).



```
 % parrot tcsh
 % cd /gsiftp/mss.ncsa.uiuc.edu/u/ac/thain
 % ls -la
total 3
drwxrwxrwx    1 thain    thain           0 Aug 26 15:00 .trash
-rwxrwxrwx    1 thain    thain       15057 Aug 26 15:00 condor.gif
-rwxrwxrwx    1 thain    thain          68 Aug 26 15:00 hello.c
-rwxrwxrwx    1 thain    thain      132921 Aug 26 15:00 lessons.pdf
 % cp lessons.pdf /tmp
 % mkdir datadir
 % vi hello.c
 % xv condor.gif
 % 
```

Figure 2. Interactive Browsing with Parrot

Improved reliability. Naturally, the networked services that are accessed by an interposition agent are far less reliable than a local filesystem. Remote services are prone to failed networks, power outages, expired credentials, and many other problems. An agent can attach an application to a service with improved reliability. For example, Rocks [37] emulates a reliable TCP connection across network outages and address changes. Parrot can also be used to add reliability at the file system layer by detecting and repairing failed I/O connections.

Private namespaces. Batch applications are frequently hardwired to use certain file names for configuration files, data libraries, and even ordinary inputs and outputs. By specifying a private namespace for each application instance, many may be run simultaneously while keeping their I/O activities separate. For example, ten instances of an application hardwired to write to *output.txt* may be redirected to write to *output.n.txt*, where n is the instance number. A private namespace may also be constructed for performance concerns. A centralized data server can only serve so many simultaneous remote applications before it becomes saturated. If copies of necessary data are available elsewhere in a system, a private namespace may be constructed to force an application to use a nearby copy. Whether for logical or performance purposes, a private namespace can be built at many points in the lifetime of an application. It may be fixed throughout the program's lifetime or it may be resolved on demand as the program runs by an external service.

Remote dynamic linking. Dynamic linking presents several problems in naming and execution. A large majority of standard applications are linked against dynamic libraries that are named and loaded at runtime. Dynamic linking reduces the use of storage and memory by allowing applications to share routines. However, this advantage can become a liability of complexity, perfectly captured by the phrase "DLL hell." Some interposition techniques permit a remotely-executing application to fetch all of its libraries from a trusted source as they are needed. Such libraries may then be shared normally at the execution site without burdening the end user.

Profiling and debugging. The vast majority of applications are designed and tested on standalone machines. A number of surprises occur when such applications are moved into a distributed system. Both the absolute and relative cost of I/O operations change, and techniques that were once acceptably inefficient (such as linear search) may become disastrously so. By attaching an interposition layer to the application, the user may easily generate a trace or summary of the I/O behavior and observe precisely what the application does. We did exactly this in a recent study of distributed applications [30].

3. Interposition Techniques Compared

There are many interposition techniques for the applications that we have described. Each has particular strengths and weaknesses. Figure 3 summarizes seven interposition techniques. They may be broken into two broad categories: internal and external. Internal techniques modify the memory space of an application process in some fashion. These techniques are flexible and efficient, but cannot be applied to arbitrary processes. External techniques capture and modify operations that are visible outside an application's address space. These techniques are less flexible and have higher overhead, but can be applied to nearly any process. The Condor project has experience building and deploying all of the internal techniques as well one external technique: the debugger trap. The remaining two external techniques we describe from relevant publications.

The simplest technique is the *polymorphic extension*. If the application structure is amenable to extension, we may simply add a new implementation of an existing interface. The user then must make small code changes to invoke the appropriate constructor or factory in order to produce the new object. This technique is used in Condor's Java Universe [32] to connect an ordinary InputStream or OutputStream to a secure remote proxy. It is also found in general purpose libraries such as SFIO. [34]

The *static library* technique involves creating a replacement for an existing library. The user is obliged to re-link the application with the new library. For example, Condor's Standard Universe [27] provides a drop-in replacement for the standard C library that provides transparent checkpointing as well as proxying of I/O back to the submission site, fully emulating the user's home environment. The *dynamic library* technique also involves creating a replacement for an existing library. However, through the use of linker controls, the user may direct the new library to be used in place of the old for any given dynamically linked library. This technique is used by DCache [12], some implementations of SOCKS [19], as well as our own Bypass [31] toolkit. The *binary rewriting* technique involves modifying the machine code of a process at runtime to redirect the flow of control. This requires very detailed knowledge of the CPU architecture in use, but this can be hidden behind an abstraction such as the Paradyn [21] toolkit. This technique has been used to "hijack" an unwitting process at runtime. [38]

Traditional debuggers make use of a specialized operating system interface for stopping, examining, and resuming a process. The *debugger trap* technique uses this interface, but instead of merely examining the process, the debugging agent traps each system call, provides an implementation, and then places the result back in the target process while nullifying the intended system call. An example of this technique is UFO [2], which allows access to HTTP and ftp resources via whole-file fetching. A difficulty with the debugger trap is that many tools compete for access to a single process' debug interface. The Tool Daemon Protocol (TDP) [22] provides an interface for managing such tools in a distributed system.

A *remote filesystem* may be used as an interposition agent by simply modifying the file server. NFS is a popular choice for this technique, and is used by the Legion [36] object-space translator, as well the Slice [4] microproxy. Finally, short of modifying the kernel itself, we may install a one-time *kernel callout* which permits a filesystem to be serviced by a user-level process. This facility can be present from the ground up in a microkernel [1], but can also be added as an afterthought, which is the case for most implementations of AFS [15].

The four internal techniques may only be applied to certain kinds of programs. Polymorphic extension and static linking only apply to those programs that can be rebuilt. The dynamic library technique requires that the replaced library be dynamic, while binary rewriting (with the Paradyn toolkit)

	internal techniques				external techniques		
	polymorphic extension	**static linking**	**dynamic linking**	**binary rewriting**	**debugger trapping**	**remote filesystem**	**kernel callout**
applies to	one library	one program	dynamic *libs*	dynamic *link*	any exc. setuid	any	any
user burden	change code	relink code	identify	identify	run command	superuser	modify os
semantic layer	fixed	any	any	any	syscall	fs ops only	syscall
init/fini	difficult	difficult	difficult	difficult	easy	impossible	easy
affect linker	no	no	no	no	yes	yes	yes
debug application	easy	easy	easy	easy	if db is child	easy	easy
secure	no	no	no	no	yes	yes	yes
hole detection	easy	difficult	difficult	difficult	easy	easy	easy
porting required	little or none	to each os revision	to each os revision	to each os revision and to each cpu	to each os	little or none	to each os

Figure 3. Properties of Interposition Techniques

requires the presence of the dynamic loader, although no particular library must be dynamic. The three external techniques apply to any process, with the exception that the debugging trap prevents the traced process from elevating its privilege level through the *setuid* feature.

The burden upon the user for each of these techniques also varies widely. For example, polymorphic extension requires small code changes while static linking requires rebuilding. These techniques may not be possible with packaged commercial software. Dynamic linking and binary rewriting require that the user understand which programs are dynamically linked and which are not. Most standard system utilities are dynamic, but many commercial packages are static. Our experience is that users are surprised and quite frustrated when an (unexpectedly) static application blithely ignores an interposition agent. The remote filesystem and kernel callout techniques impose the smallest user burden, but require a cooperative system administrator to make the necessary changes. The debugger trap imposes a small burden on the user to simply invoke the agent executable.

Perhaps the most significant difference between the techniques is the ability to trap different layers of software. Each of the internal techniques may be applied at any layer of code. For example, Bypass has been used to instrument an application's calls to the standard memory allocator, the X Window System library, and the OpenGL library. In contrast, the external techniques are fixed to particular interfaces. The debugger trap only operates on physical system calls, while the remote filesystem and kernel callout are limited to certain filesystem operations.

Differences in these techniques affect the design of code that they attach to. Consider the matter of implementing a directory listing on a remote device. The internal techniques are capable of intercepting library calls such as *open* and *opendir*. These are easily mapped to remote file access protocols, which generally have separate procedures for accessing files and directories. However, the Unix interface unifies files and directories; both are accessed through the system call *open*. External techniques must accept an *open* on either a file or directory and defer the binding to a remote operation until either *read* or *getdents* is invoked. The choice of interposition layer affects the design of the agent.

The external techniques also differ in the range of operations that they are able to trap. While the debugger trap can modify any system call, the remote filesystem and kernel callout techniques are limited to filesystem operations. A particular remote filesystem may have even further restrictions. For example, the stateless NFS protocol has no representation of the system calls *open* and *close*. Without access to this information, the interposed service cannot provide semantics significantly different than those provided by NFS. Further, such file system interfaces do not express any binding between individual operations and the processes that initiate them. That is, a remote filesystem agent sees a *read* or *write* but not the process id that issued it. Without this information, it is difficult or impossible to performing accounting for the purposes of security or performance.

A number of important activities take place during the initialization and finalization of a process: dynamic libraries are loaded; constructors, destructors, and other automatic routines are run; I/O streams are created or flushed. During these transitions, the libraries and other resources in use by a process are in a state of flux. This complicates the implementation of internal agents that wish to intercept such activity. For example, the application may perform I/O in a global constructor or destructor. Thus, an internal agent itself cannot rely on global constructors or destructors: there is no ordering enforced between those of the application and those of the agent. Likewise, a dynamically loaded agent cannot interpose on the actions of the dynamic linker. The programmer of such agents must not only exercise care in constructing the agent, but also in selecting the libraries invoked by the agent. Such code is time consuming to create and debug. These activities are much more easily manipulated through external techniques. For example, external techniques can easily trap and modify the activities of the dynamic linker.

	getpid	stat	open/close	read 1B	read 8KB	write 1B	write 8KB	bandwidth
unmod	.18±.03 μs	1.85±.09	3.18±.08	.93± .23	3.27±.19	2.77± .05	6.92±.17	282±13 *MB/s*
rewrite	.21±.25 μs	1.82±.02	3.21±.05	.95± .02	3.26±.03	2.58± .17	6.70±.05	280± 7 *MB/s*
static	.21±.02 μs	1.80±.17	3.59±.05	.96± .03	3.34±.02	2.64± .03	6.71±.21	280±17 *MB/s*
dynamic	1.22±.01 μs	3.60±.10	5.53±.06	2.00± .08	4.31±.09	3.92±1.07	7.95±.21	278± 4 *MB/s*
(α unmod)	(6.8x)	(1.9x)	(1.7x)	(2.2x)	(1.3x)	(1.4x)	(1.1x)	(0.99x)
debug	10.06±.21 μs	55.41±.50	42.09±.06	19.38±1.03	30.99±.26	27.69± .20	44.02±.29	122± 4 *MB/s*
(α unmod)	(56x)	(30x)	(13x)	(21x)	(9x)	(10x)	(6x)	(0.43x)

Figure 4. Overhead of Interposition Techniques

No code is ever complete nor fully debugged. Production deployment of interposition agents requires that users be permitted to debug both applications and agents. All techniques admit debugging of user programs, with the only complication arising in the debugger trap. For obvious reasons, a single process cannot be debugged by two processes at once, so a debugger cannot be attached to an instrumented process. However, a debugger trap agent can be used to manage an entire process tree, so instead the user may use the agent to invoke the debugger, which may then invoke the application. The debugger's operations may be trapped just like any other system call and passed along to the application, all under the supervision of the agent.

Interposition agents may be used for security as well as convenience. An agent may provide a *sandbox* which prevents an untrusted application from modifying any external data that it is not permitted to access. The internal techniques are not suitable for this security purpose, because they may easily be subverted by a program that invokes system calls directly without passing through libraries. The external techniques, however, cannot be fooled in this way and are thus suitable for security.

Related to security is the matter of *hole detection*. An interposition agent may fail to trap an operation attempted by an application. This may simply be a bug in the agent, or it may be that the interface has evolved over time, and the application is using a deprecated or newly added interface that the agent is not aware of. Internal agents are especially sensitive to this bug. As standard libraries develop, interfaces are added and deleted, and modified library routines may invoke system calls directly without passing through the corresponding public interface function. For example, *fopen* may invoke the *open* system call without passing through the *open* function. Such an event causes general chaos in both the application and agent, often resulting in crashes or (worse) silent output errors. No such problem occurs in external agents. Although interfaces still change, any unexpected event is detected as an unknown system call. The agent may then terminate the application and indicate the exact problem.

The problem of hole detection must not be underestimated. Our experience is that any significant operating system upgrade includes changes to the standard libraries, which in turn require modifications to internal trapping techniques. Thus, internal agents are rarely forward compatible. Further, identifying and fixing such holes is time consuming. Because the missed operation itself is unknown, one must spend long hours with a debugger to see where the expected course of the application differs from the actual behavior. Once discovered, a new entry point must be added to the agent. The treatment is simple but the diagnosis is difficult. We have learned this lesson the hard way by porting both the Condor remote system call library and the Bypass toolkit to a wide variety of Unix-like platforms.

For these reasons, we have described *porting* in Figure 3 as follows. The polymorphic extension and the remote filesystem require little or no effort to build on a new system. The debugger trap and the kernel callout have significant system dependent components to be ported to each operating system, but the nature and stability of these interfaces make this a tractable task. The remaining three techniques – static linking, dynamic linking, and binary rewriting – should be viewed as a significant porting challenge that must be revisited at every minor operating system upgrade.

Figure 4 compares the performance of four transparent interposition techniques. We constructed a benchmark C program which timed 100,000 iterations of various system calls on a 1545 MHz Athlon XP1800 running Linux 2.4.18. Available bandwidth was measured by reading a 100 MB file sequentially in 1 MB blocks. The mean and standard deviation of 1000 cycles of each benchmark are shown. File operations were performed on an existing file in a temporary file system. The *unmod* case gives the performance of this benchmark without any agent attached, while the remaining five show the same benchmark modified by each interposition technique. In each case, we constructed a very minimal agent to trap system calls and invoke them without modification.

As can be seen, the binary rewriting and static linking methods add no significant cost to the application. The dynamic method has overhead on the order of microseconds, as it must manage the structure of (potentially) multiple agents and invoke a function pointer. However, these overheads are quickly dominated by the cost of moving data in and out of the process. The debugger trap has the greatest overhead of all the techniques, ranging from a 56x slowdown for getpid to a 6x slowdown for writing 8 KB. Most importantly, the

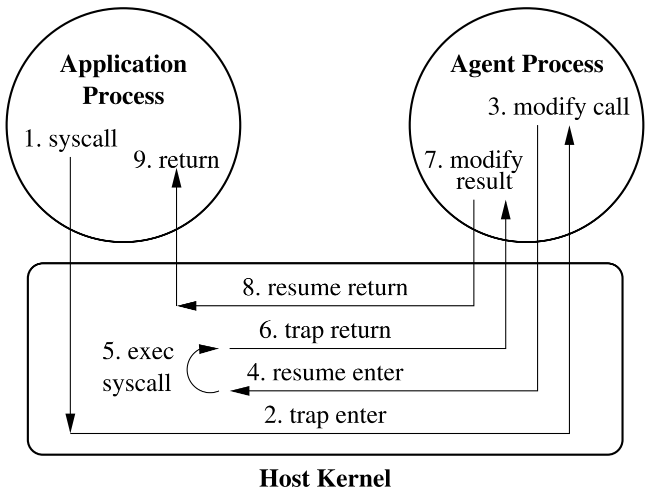


Figure 5. Ptrace Control Flow

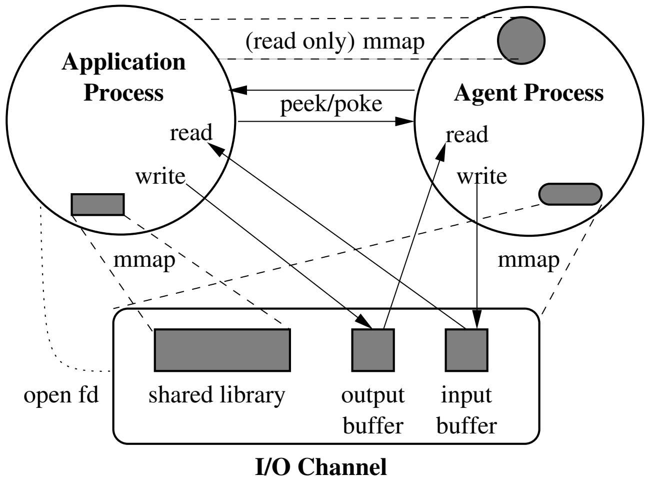


Figure 6. Ptrace Data Flow

bandwidth measurement demonstrates that the debugger trap achieves less than half of the unmodified I/O bandwidth. It should be fairly noted that this latency and bandwidth will be dominated by the latency and bandwidth of accessing remote services on commodity networks. Security and reliability come at a measurable cost; the reasons for this cost are explained in the next section.

4. The Debugger Trap in Detail

Here, we concentrate on the details of the debugger trap. Other methods are explained in some detail in the publications referenced above. Alexandrov et al. [2] have described the use of the Solaris *proc* debugger trap to instrument a process in this manner. However, Linux is currently a much more widely deployed platform for scientific and distributed computing. Its *ptrace* debugger model is generally considered inferior to the Solaris *proc* model; it can still be used for interposition, but it has limitations that must be accommodated.

Figure 5 shows the control flow necessary to trap a system call through the *ptrace* interface. The agent process registers its interest in an application process with the host kernel. At each attempt by the application to invoke a system call, the host kernel notifies the agent of the attempt. The agent may then modify the application's address space or registers, including the system call and its arguments. Once satisfied, the agent instructs the host kernel to resume the system call. At completion, the agent is given another opportunity to modify the application and the result. Once satisfied, the agent resumes the return from the system call, and the application regains control. This large number of context switches accounts for the high latency measured in Figure 4.

Although conceptually simple, there are two complexities in the *ptrace* interface:

Process ancestry. The *ptrace* interface forces all traced processes to become the immediate children of the tracing processes. This is because notification of trace events occurs through the same path as notification of child completion events: the tracing process receives a signal, and then must call *waitpid* to retrieve the details. As a consequence, any tracing tool that wishes to follow a tree of processes must maintain a table of process ancestry. All system calls that communicate information about children (such as *waitpid*) must be trapped and emulated by the agent. If a traced process forks, the Linux kernel (inexplicably) does not propagate the tracing flags to the child. This may be overcome by trapping instances of *fork* and converting them into the more flexible (and Linux specific) *clone* system call, which can be instructed to create a new process with tracing activated.

Data flow. The emulation of system calls requires the ability to move data in and out of the target application. Figure 6 shows all of the necessary data flow techniques. The most convenient is to access a special file (*/proc/n/mem*) that represents the entire memory space of the application. This can be modified with *read* and *write*, or can be mapped into the address space of the agent process. Although this provides high-bandwidth read access, writing to this file is not permitted. [2] A pair of *ptrace* calls, *peek* and *poke*, are provided to read or write a single word in the target application. This interface can be used for moving small amounts of data into the target application, but is obviously not suited for moving large amounts of data such as is required by the *read* and *write* system calls.

To move data efficiently, the application must be coerced

[2]Writing to this file has been implemented, but is commented out in the kernel source. The reasons appear to be lost to folklore, although comments in the source suggest security concerns. Clearly, both read and write access to another process's address space must be revoked if the target process can raise its privilege level via setuid. It is not clear to what extent such revocation is implemented.

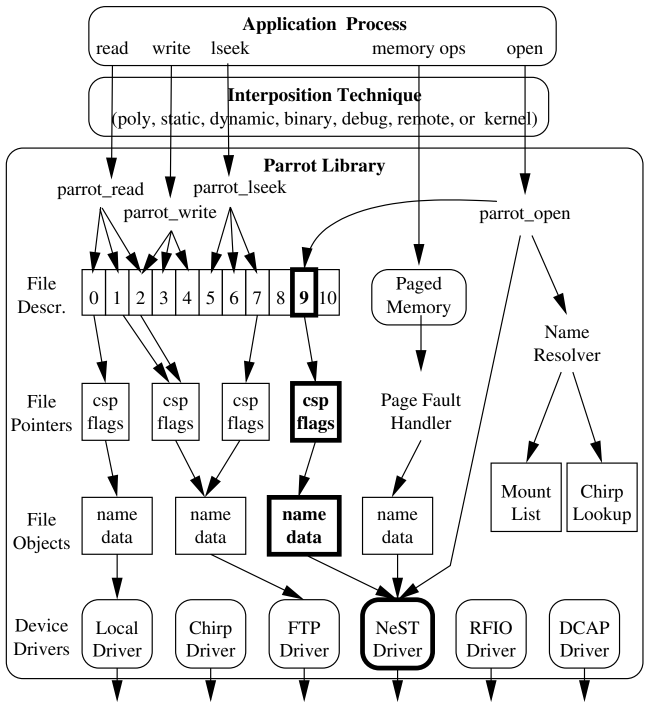


Figure 7. Architecture of Parrot

into assisting the agent. This is accomplished by converting many system calls into *preads* and *pwrites* on a shared buffer called the *I/O channel*. This is an ordinary file, created by the agent, passed implicitly, and shared among all of its children. The agent maps the channel into memory, to minimize copying, while all of the application processes simply maintain a file descriptor pointing to the channel.

For example, suppose that the application issues a *read* on a remote file. Upon trapping the system call entry, the agent examines the parameters of *read* and retrieves the needed data. These are copied directly into a buffer in the channel. The *read* is then modified (via *poke*) to be a *pread* that accesses the I/O channel instead. The system call is resumed, and the application pulls in the data from the channel, unaware of the activity necessary to place it there. This data copy accounts for the diminished bandwidth in Figure 4.

5. Architecture of Parrot

Parrot is a library for performing POSIX-like I/O on remote data services. It provides an interface with entry points like *parrot_open* and *parrot_read*. An application may be written or modified to invoke the library directly, or it may be attached via the various interposition techniques described above. The internal structures of Parrot, shown in Figure 7, bear a strong resemblance to an operating system. Parrot tracks a family tree of child processes, for each recording a table of open file descriptors, seek pointers, and similar device-independent structures. At the lowest layer are a series of device drivers that implement access to remote I/O system. Unlike an operating system, Parrot does not know the structure of remote devices at the level of inodes and blocks. It refers to remote open files by name, and may multiplex many applications' I/O requests through one remote channel.

Parrot has a large number of entry points for I/O operations. We may classify them into two categories: operations on file descriptors and operations on file names. The former traverse most of the data structures in Parrot, 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 Parrot, these commands check the validity of the arguments, and then descend the various data structures. *read* and *write* examine the current 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 Parrot, these commands first pass through the *name resolver*, which may transform the program-supplied name(s) according to a variety of rules and systems. The transformed names are passed directly to the device driver, which performs the operation on the remote system.

Name resolution itself is served by several drivers. In the simplest case, no mapping is present, and Parrot operates on the name unchanged. Applications may specify ordinary local file names such as */etc/passwd* or fully-qualified remote filenames such as */ftp/ftp.cs.wisc.edu/RoadMap*. A **mount list** driver makes use of a simple file that maps logical file names and directories to remote file names, much like the Unix *fstab*. Alternatively, the **chirp** I/O driver itself provides a *lookup* RPC that permits a remote controller to perform run-time name binding on behalf of an application.

The name resolver is a natural place for attaching an application to other external naming systems. For example, both the Replica Location Service [33] and the Handle System [28] resolve abstract document names into physical locations where they may be accessed. The same interface could be used to employ Parrot as a sandbox, selectively permitting, renaming, or rejecting access to certain files in a manner similar to that of Janus [14].

Most POSIX 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. Parrot accomplishes the same thing using its own underlying drivers, thus reducing memory mapped files into the same mechanisms as other open files.

Memory-mapped files are supported in one of two ways, depending on the interposition method in use. If Parrot is attached via an internal technique, then memory mapped files may be supported by simply allocating memory with *malloc* and loading the necessary data into memory by invoking the necessary device driver. As a matter of policy, the entire file can be loaded when *mmap* is invoked, or it can be paged in on demand by setting the necessary memory protections and trapping the software interrupts generated by access to that memory. If Parrot is attached via an external technique, then the entire file is loaded into the I/O channel, and the application is redirected to *mmap* that portion of the channel, as shown in Figure 6. Parrot does not currently have any write mechanism or policy for memory-mapped files, as we have yet to encounter any application that requires it.

Parrot has two buffering disciplines. By default, Parrot simply performs fine-grained partial file operations on remote services to access the minimal amount of data to satisfy an application's immediate *read*s and *write*s. We have taken this route for several reasons. First, whole-file fetching introduces a large latency when a file is first opened. This is often 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; we do not wish to introduce a new synchronization problem. Finally, a variety of systems have already been proposed for managing wide area replicated data [9, 25, 6]. We prefer to make Parrot leverage such systems (via fine-grained access protocols) rather than implement replica management anew.

Optionally, Parrot may perform whole-file staging and caching upon first *open*, similar to that of UFO [2]. Once this long latency is paid, a file may be accessed efficiently in local storage. Protocols that only provide sequential access, such as ftp, require the use of the cache to implement random access. At each *open*, a cached file is validated by performing a remote *stat* (or its equivalent, described below.) If the file's size or modification time has changed, then it is re-fetched.

6 Protocols and Semantics

Parrot is equipped with a variety of drivers for communicating with external storage systems; each has particular features and limitations. The simplest is the **Local** driver, which simply passes operations on to the underlying operating system. The **Chirp** protocol was designed by the authors in an earlier work [32] to provide remote I/O with semantics very similar to POSIX. A standalone chirp server is distributed with Parrot. The venerable **File Transfer Protocol (FTP)** has been in heavy use since the early days of the Internet. Its simplicity allows for a wide variety of of implementations, which, for our purposes, results in an unfortunate degree of imprecision which we will expand upon below. Parrot supports the secure GSI [3] variant of ftp. The **NeST** protocol is the native language of the NeST storage appliance [8], which provides an array of authentication, allocation, and accounting mechanisms for storage that may be shared among multiple transient users. The **RFIO** and **DCAP** protocols were designed in the high-energy physics community to provide access to hierarchical mass storage devices such as Castor [5] and DCache [12].

Because Parrot must preserve POSIX semantics for the sake of the application, our foremost concern is the ability of each of these protocols to provide the necessary semantics. Performance is a secondary concern, although it is affect significantly by semantic issues. A summary of the semantics of each of these protocols is given in Figure 8.

In POSIX, name binding is based on a separation between the namespace of a filesystem and the file objects (i.e. inodes) that it contains. The *open* system call performs an atomic binding of a file name to a file object, which allows a program to lock a file object independently of the renaming, linking, or unlinking of names that point to it. This model is reflected in the Chirp, RFIO, and DCAP protocols, which all provide distinct *open/close* actions separately from data access. FTP and NeST have a *get/put* model, performing a name lookup at every data access. In this model, an application may lose files it has open if they are manipulated by another process.

The distinction between the two models begins to blur if we consider recovery of a failed connection. The *get/put* models have the advantage of statelessness. If Parrot loses a connection to such a service, it merely re-establishes the connection and does no further work. However, if a connection is lost to an *open/close* service, Parrot must reconstruct the state by re-opening the necessary files. Of course, this recovery procedure may reconnect to a different file object than was referenced by the original open. To detect this, Parrot examines the remote inode when opening and re-opening remote files in order to detect if the binding has changed in the mean time. If it has, then the file descriptor is considered stale and all further reads and writes on that descriptor fail with the error *ESTALE*, much as in NFS.

With the exception of FTP, all of the protocols provide inexpensive random (i.e. non-sequential) access to a file without closing and re-opening it. This permits the efficient manipulation of a small portion of a large remote file without retrieving the whole thing. The sequential nature of FTP requires that Parrot make local copies of such files in order to make changes and then replace the whole file.

Directories are supported completely by Chirp, NeST, and RFIO; one may create, delete and list their contents. DCAP does not currently support directory access, although this may be added in a later version. (This is because DCAP is typically used alongside an kernel NFS client for metadata access.) Support for directories in FTP varies greatly. Although the FTP standard mandates two distinct commands for directory lists, LIST and NLST, there is little agreement on their proper behavior. LIST provides a completely free-

	name binding	discipline	dirs	metadata	symlinks	connections
posix	open/close	random	yes	direct	yes	-
chirp	open/close	random	yes	direct	yes	per client
ftp	get/put	sequential	varies	indirect	no	per file
nest	get/put	random	yes	indirect	yes	per client
rfio	open/close	random	yes	direct	no	per file/op
dcap	open/close	random	no	direct	no	per client

Figure 8. Protocol Compatibility with POSIX

form text dump that is readable to humans, but has no standard machine-readable structure. NLST is meant to provide a simple machine-readable list of directory entries, but we have encountered servers that omit subdirectory names, some that omit names beginning with dot (.), some that insert messages into the directory list, and even some that do not distinguish between empty and non-existent directories.

Most metadata is communicated in the POSIX interface through the *stat* structure returned by the *stat*, *fstat*, and *lstat* system calls. Chirp, RFIO, and DCAP all provide direct single RPCs that fill this structure with the necessary details. FTP and NeST do not have single calls that provide all this information, however, the necessary details may be obtained through multiple RPCs that determine the type, size, and other details one by one.

Only Chirp and NeST provide support for managing symbolic links. This feature might be done without, except that remote I/O protocols are often used to expose existing filesystems that already contain symbolic links. This can result in confusion interactions for programs, as well as the end user. For example, a symbolic link may appear as in a directory listing, but without explicit operations for examining links, it will appear to be an inaccessible file with unusual access permissions.

Finally, the connection structure of a remote I/O protocol has implications for semantics as well as performance. Chirp, NeST, and DCAP require one TCP connection between each client and server. FTP and RFIO require a new connection made for each file opened. In addition, RFIO requires a new connection for each operation performed on a non-open file. Because most file system operations are metadata queries, this can result in an extraordinary number of connections in a short amount of time. Ignoring the latency penalties of this activity, a large number of TCP connections can consume resources at clients, servers, and network devices such as address translators.

7. Errors and Boundary Conditions

Error handling has not been a pervasive problem in the design of traditional operating systems. As new models of file interaction have developed, attending error modes have been added to existing systems by expanding the software interface at every level. For example, the addition of distributed file systems 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 an interposition agent. Applications use the existing interface, and we have neither the desire nor the ability to change it. Sometimes, if we are lucky, we may re-use an error such as *ESTALE* for an analogous, if not identical purpose. Yet, the underlying device drivers generate errors ranging from the vague "file system error" to the microscopically precise "server's certification authority is not trusted." How should the unlimited space of errors in the lower layers be transformed into the fixed space of errors available to the application?

Before we answer this question, we must remind ourselves that an interposition agent does not live in a vacuum, nor is it the last line of defense for errors. As Figure 1 shows, the application and the agent often work under the supervision of a batch system. In this context, we may appeal to the batch system to take some higher-level scheduling action. This is not to say that we should always pass the buck to the batch system. Rather, we must perform triage:

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

Each of the three types of errors – transformable, permanent, and transient – come from two distinct sources of errors – a mismatch of requests, or a mismatch of results. 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.

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 *getdents*, *lseek*, and *stat*.

Some I/O services, such as DCAP, do not permit directory listings. A call to *getdents* cannot possibly succeed. Such a failure may be trivially represented to the calling application as "permission denied" or "not implemented" without undue confusion. Applications understand that *getdents* 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 non-terminal file 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 without local caching enabled, 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 with a 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.

One might cause *stat* to report "permission denied" on such systems, under the assumption that brutal honesty is best. Unfortunately, this causes all but the most trivial of programs to fail. *stat* is a very frequent operation that is called implicitly by all manner of code, including command-line tools, large applications, and the standard C library. At first glance, it appears that the necessary information simply cannot be extracted from most remote I/O systems. However, we may construct a workaround by surveying the actual uses of *stat*:

- 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.

- 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.

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

- Unique identity. Command line tools use the unique device and file numbers returned by *stat* to determine if two file names refer to the same physical file. This is used to prevent accidental overwriting and recursive operations.

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 for both reading and writing. Each device driver fills in whatever values in the structure it is able to determine, perhaps using multiple remote operations, and then fills the rest with defaults. For example, an inode may be computed from a hash of the file's full path, while the last modification time may be set to the current time. Or course, if the device driver can determine that the file actually does not exist, then it may truthfully cause *stat* to fail.

Of particular interest is the block size field returned by *stat*. In practice, the physical block size of the underlying device is irrelevant to the file abstraction; some devices may not have the concept of a block at all. However, many routines – particularly the standard C library – use the block size as an indication of the file's optimal transfer size. Parrot leverages this interpretation to hide the (potentially) high latency of both interposition and remote access. By default, Parrot indicates a block size of one megabyte for all files. We explain the reason for this choice below.

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 an expert system or the functional error-interview system described by Efe [11]. Figure 9 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 move to it. If that succeeds, the error is "not a file." Otherwise, we attempt to SIZE 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 increases 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 if the remote filesystem is simultaneously modified by another process.

There is also a very large space of infrequent errors that simply have no expression at all in the application's inter-

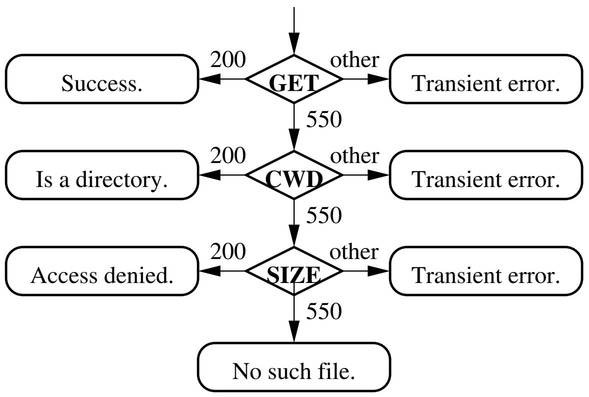


Figure 9. An Error Interview

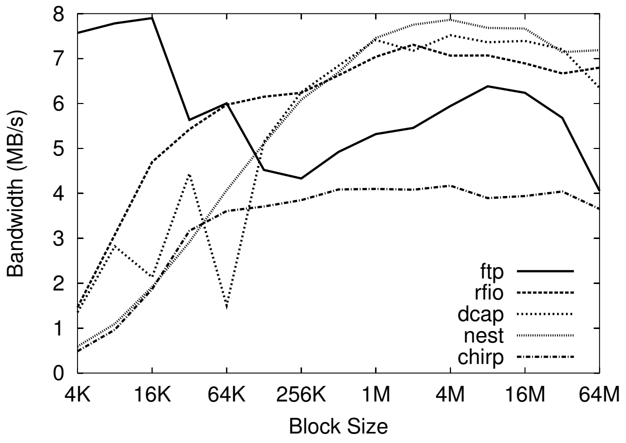


Figure 10. Throughput of 128 MB File Copy

face. A NeST might declare that a disk allocation has expired and been deleted. An FTP server may respond that a backing store is temporarily offline. User credentials, such as Kerberos or GSI certificates, may expire, and no longer be valid. In response, we may reallocate lots, rebuild connections, or attempt to renew certificates. 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 occurred. Because these errors have no analogue in the interface to the application, Parrot must declare a transient error and seek to reschedule the job.

8. Integrating Computation and I/O

The reality of transient errors requires the integration of computation and storage systems. An interposition agent is a natural device for mediating these systems, because it is able to take complex actions on either without involving the application in the flow of control.

Generally speaking, an agent is limited to the simple scheduling actions exposed by the POSIX interface. For example, a permanent error is caused by forcing the process to call *exit(1)*, indicating that it has completed unsuccessfully and should be returned to the submitter. A transient error is indicated by terminating the process with a forcible kill signal. A batch system such as Condor interprets this as evidence of outside interference, comparable to a workstation owner evicting a visiting process. The job is then placed elsewhere by the batch scheduler.

Here we must emphasize the difference between local (operating system) scheduling and batch scheduling. In response to a transient error, an agent could simply block 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 scheduler. However, what the process is actually doing in the local scheduler is irrelevant to the batch scheduler. Unless the program (or agent) issues some explicit instruction to the batch system, it still is in possession of the CPU. It will continued to be charged for holding the resource, regardless of whether it is consuming physical cycles.

If the batch system is Condor, more complex information can be attached to a transient error through the use of a control port at the execution site. Each job executing in a Condor system is overseen by a process known as a *starter*. One of the starter's tasks is to provide a local control port for an application to invoke complex actions within the batch system. (Incidentally, the protocol spoken on this port is Chirp.) A *constrain* RPC provided by the starter specifies a clause to be added to the job's scheduling constraints expressed in the ClassAd [24] language. For example, if Parrot discovers that a needed file is not scheduled to be staged in from tape until 9:15 AM, it can use *constrain* to add the requirement *(DayTime() > '9:15:00')* and then kill the application. Condor will not re-place the process until the new requirements are satisfied. Alternatively, suppose that Parrot has already moved a dataset to a nearby cache at considerable cost. Without releasing the CPU, it might call *constrain* to add the requirement *(Subnet=="128.105.175")* and then continue processing. If the application should fail or be evicted at a later time, it will be re-placed by Condor at any machine satisfying the constraints: that is, on the same subnet as the needed data. Multiple calls to *constrain* overwrite the previous constraint, so that previous decisions may be un-done.

The notion of job-directed resource management is introduced in J. Pruyne's doctoral thesis [23]. A resource management interface called CARMI permits a running job to request and release external resources at run-time. A similar idea is found in the notion of execution domains [7], where the Condor shadow directs future allocations based on the location of checkpoint images. The Chirp *constrain* facility combines both of these ideas by permitting the job (or its agent) to direct further allocation requests in concert with the state of the external system.

protocol	**server**	**stat**	**open/close**	**read 1B**	**read 8KB**	**write 1B**	**write 8KB**	**bandwidth**
chirp	(chirp-0.9.7)	.50± .14 *ms*	.84± .09	.61± .04	2.80± .06	.38± .03	2.23± .04	4.1 *MB/s*
ftp	(wu-ftpd-2.6.2)	.87± .09 *ms*	2.82± .26	*(no random partial-file access)*				7.9 *MB/s*
nest	(nest-0.9.3)	2.51± .05 *ms*	2.53± .17	2.96± .17	4.48± .14	5.53± .09	7.41± .32	7.9 *MB/s*
rfio	(rfiod-1.5.2.3)	13.41± .28 *ms*	23.11± 1.29	.50± .06	3.32± .14	39.80±1.32	2.85± .18	7.3 *MB/s*
dcap	(dcache-1.2.25)	152.53±16.68 *ms*	159.09±16.68	40.05±0.17	3.01±0.62	40.14± .59	3.14± .62	7.5 *MB/s*

Figure 11. Performance of I/O Protocols On a Local-Area Network

9. Performance

We have deferred a discussion of performance until this point so that we may see the performance effects of semantic constraints. Although it is possible to write applications explicitly to use remote I/O protocols in the most efficient manner, Parrot must provide conservative and complete implementations of POSIX operations. For example, an application may only need to know the size of a file, but if it requests this information via *stat*, Parrot is obliged to fill the structure with everything it can, possibly at great cost.

The I/O services discussed here, with the exception of Chirp, are designed primarily for efficient high-volume data movement. This is demonstrated by Figure 10, which compares the throughput of the protocols at various block sizes. The throughput was measured by copying a 128 MB file into the remote storage device with the standard *cp* command equipped with Parrot and a varying default block size, as controlled through the *stat* emulation described above.

Of course, the absolute values are an artifact of our system, however, it can be seen that any of the protocols can be tuned to near optimal performance for mass data movement. (The default block size, explained earlier, is chosen to be 1 MB as a compromise between all protocols.) The exception is Chirp, which only reaches about one half of the available bandwidth. This is because of the strict RPC nature required for POSIX semantics; the Chirp server does not extract from the underlying filesystem any more data than necessary to supply the immediate read. Although it is technically feasible for the server to read ahead in anticipation of the next operation, such data pulled into the server's address space might be invalidated by other actors on the file in the meantime and is thus semantically incorrect.

The hiccup in throughput of DCAP at a block size of 64KB is an unintended interaction with the default TCP buffer size of 64 KB. The developers of DCAP are aware of the artifact and recommend changing either the block size of the buffer size to avoid it. This is reasonable advice, given that all of the protocols require tuning of some kind.

Figure 11 benchmarks the latency of POSIX-equivalent operations in each I/O protocol. These measurements were obtained in a manner identical to that of Figure 4, with the indicated servers residing on the same system as in Figure 10. Notice that the latencies are measured in milliseconds, whereas Figure 4 gave microseconds.

We hasten to note that this comparison, in a certain sense, is not "fair." These data servers provide vastly different services, so the performance differences demonstrate the cost of the service, not the cleverness of the implementation. For example, Chirp and FTP achieve low latencies because they are lightweight translation layers over an ordinary file system. NeST has somewhat higher latency because it provides the abstraction of a virtual file system, user namespace, access control lists, and a storage allocation system, all built on an existing filesystem. The cost is due to the necessary metadata log that records all such activity that cannot be stored directly in the underlying file system. Both RFIO and DCAP are designed to interact with mass storage systems; single operations may result in gigabytes of activity within a disk cache, possibly moving files to or from tape. In that context, low latency is not a concern.

That said, several things may be observed from this table. Although FTP has benefited from years of optimizations, the cost of a *stat* is greater than that of Chirp because of the need for multiple round trips to fill in the necessary details. The additional latency of *open/close* is due to the multiple round trips to name and establish a new TCP connection. Both RFIO and DCAP have higher latencies for single byte reads and writes than for 8KB reads and writes. This is due to buffering which delays small operations in anticipation of further data. Most importantly, all of these remote operations exceed the latency of the debugger trap itself by several orders of magnitude. Thus, we are comfortable with the previous decision to sacrifice performance in favor of reliability in the interposition technique.

We conclude with a macrobenchmark similar to the Andrew benchmark. [15] [3] This Andrew-like benchmark consists of a series of operations on the Parrot source tree, which consists of 13 directories and 296 files totaling 955 KB. To prepare, the source tree is moved to the remote device. In the **copy** stage, the tree is duplicated on the remote device. In the **list** stage, a detailed list (ls -lR) of the tree is made. In the **scan** stage, all files in the tree are searched (grep) for a text string. In the **make** stage, the software is built. From an I/O perspective, this involves a sequential read of every source file, a sequential write of every object file, and a series of random reads and writes to create the executables. In

[3]We considered the original Andrew benchmark, however, it is quite small by today's standards and has aged to the point where it no longer compiles with standard tools.

distance	cache	protocol	copy	list	scan	make	delete
local	off	local	.15± .02 *sec*	.09± .20	.08± .02	65.38±3.47	.86± .18 *sec*
local	off	chirp	1.22± .03 *sec*	.34± .02	.40± .01	81.02±1.46	.79± .01 *sec*
lan	off	chirp	6.16± .22 *sec*	.57± .30	1.32± .03	144.00±1.35	1.26± .02 *sec*
lan	on	chirp	10.67± .90 *sec*	.53± .07	4.72± .32	95.05±2.33	1.24± .03 *sec*
lan	on	ftp	34.88±1.72 *sec*	1.47± .02	17.78±1.14	122.54±3.14	2.95± .15 *sec*
lan	on	nest	52.35±4.18 *sec*	12.92±4.87	28.14±4.52	307.19±3.26	31.73±4.37 *sec*
lan	on	rfio			*(overwhelmed by repeated connections)*		
lan	on	dcap			*(does not support directories without nfs)*		

Figure 12. Performance of the Andrew-Like Benchmark

the **delete** stage, the tree is deleted.

Figure 12 compares the performance of the Andrew-like benchmark in a variety of configurations. In the three cases above the horizontal rule, we measure the cost of each layer of software added: first with Parrot only, then with a Chirp server on the same host, then with a Chirp server across the local area network. Not surprisingly, the I/O cost of separating computation from storage is high. Copying data is much slower over the network, although the slowdown in the make stage is quite acceptable if we intend to increase throughput via remote parallelization.

In the two cases adjacent to the rule, the only change is the enabling of caching. As might be expected, the cost of unnecessary duplication causes an increase in copying the source tree, although the difference is easily made up in the make stage, where the cache eliminates the multiple random I/O necessary to link executables. The list and delete stages only involve directory structure and metadata access and are thus not affected by the cache.

In the five cases below the horizontal rule, we explore the use of various protocols to run the benchmark. In all of these cases, caching is enabled in order to eliminate the cost of random access as discussed. The DCAP protocol is semantically unable to run the benchmark, as it does not provide the necessary access to directories. The RFIO protocol is semantically able to run the benchmark, but the high frequency of filesystem operations results in a large number of TCP connections, which quickly exhausts networking resources at both the client and the server, thus preventing the benchmark from running. Chirp, FTP, and NeST are all able to complete the benchmark. The NeST results have a high variance, due to delays incurred while the metadata log is periodically compressed. The difference in performance between Chirp, FTP, and NeST is primarily attributable to the cost of metadata lookups. All the stages make heavy use of *stat*; the multiple round trips necessary to implement this completely for FTP and NeST have a striking cumulative effect.

10. Conclusions

Interposition agents provide a stable platform for bringing old applications into new environments. We have outlined the difficulties that we have encountered as well as the solutions we have constructed in the course of building and deploying several types of agents within the Condor project. In general, the interposition techniques with the lowest overhead require the greatest amount of knowledge about the application, while the more expensive techniques are more reliable but less flexible. As we have shown, the Linux debugger trap has several limitations, but can still be put to good use. As interest grows in the use of virtual machines in distributed systems [35] the need for powerful but low overhead methods of interposition grows. The appropriate interface for this task is still an open research topic.

The notion of virtualizing or multiplexing an existing interface is a common technique [18, 10], but the plague of errors and other boundary conditions seems to be suffered silently by practitioners. Such problems are rarely publicized, however, we are aware of two excellent exceptions. C. Metz [20] describes how the Berkeley sockets interface is surprisingly hard to multiplex. T. Garfinkel [13] describes the subtle semantic problems of sandboxing untrusted applications.

We have emphasized that the problem of error handling forces the integration of access to computation and storage. Although sensible partitioning of technical problems tends to focus practitioners on one problem or the other, a holistic view is needed to properly bring the two together. Narrow interfaces such as *exit* and *kill* are widely portable but provide very coarse-grained control. We have presented an example of a more powerful interface: the Chirp *constrain* call. We believe a richer interface for integration remains a a fruitful area of research.

For more information about Parrot:
`http://www.cs.wisc.edu/~thain/research/parrot`

11. Acknowledgments

We thank John Bent and Sander Klous for their assistance in deploying and debugging Parrot. Victor Zandy wrote the mechanism for binary rewriting. Alain Roy gave thoughtful comments on early drafts of this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer Technical Conference*, Atlanta, GA, 1986.

[2] 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.

[3] 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*, pages 161–163, 2000.

[4] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[5] O. Barring, J. Baud, and J. Durand. CASTOR project status. In *Proceedings of Computing in High Energy Physics*, Padua, Italy, 2000.

[6] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[7] J. Basney, M. Livny, and P. Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. *Computer Physics Communications*, 140, 2001.

[8] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-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. In *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 1987.

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

[12] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman. dCache, a distributed storage data caching system. In *Proceedings of Computing in High Energy Physics*, Beijing, China, 2001.

[13] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, 1996.

[15] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

[17] M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

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

[19] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS protocol version 5. Internet Engineering Task Force (IETF) Request for Comments (RFC) 1928, March 1996.

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

[21] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[22] B. Miller, A. Cortes, M. A. Senar, and M. Livny. The tool daemon protocol (TDP). In *Proceedings of Supercomputing*, Phoenix, AZ, November 2003.

[23] J. Pruyne. *Resource Management Services for Parallel Applications*. PhD thesis, University of Wisconsin-Madison, 1996.

[24] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, October 2000.

[25] A. Samar and H. Stockinger. Grid Data Management Pilot. In *In Proceedings of IASTED International Conference on Applied Informatics*, Innsbruck, Austria, February 2001.

[26] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Technical Conference*, pages 41–54, 1996.

[27] M. Solomon and M. Litzkow. Supporting checkpointing and process migration outside the Unix kernel. In *Proceedings of the USENIX Winter Technical Conference*, pages 283–290, 1992.

[28] S. Sun. Establishing persistent identity using the handle system. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[29] 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*, pages 325–333, San Francisco, California, August 2001.

[30] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, WA, June 2003.

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

[32] D. Thain and M. Livny. Error scope on a computational grid. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, July 2002.

[33] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid*, May 2001.

[34] K.-P. Vo. The discipline and method architecture for reusable libraries. *Software: Practice and Experience*, 30:107–128, 2000.

[35] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[36] B. White, A. Grimshaw, and A. Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, August 2000.

[37] V. Zandy and B. Miller. Reliable network connections. In *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, GA, September 2002.

[38] V. Zandy, B. Miller, and M. Livny. Process hijacking. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.