Scalable Computing: Practice and Experience
Volume 6, Number 3, pp. 9–18. http://www.scpe.org

ISSN
© 2005 SWPS

PARROT: AN APPLICATION ENVIRONMENT FOR DATA-INTENSIVE COMPUTING

DOUGLAS THAIN AND MIRON LIVNY*

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 middleware 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 and select one method, the debugger trap, that is the slowest but also the most reliable. 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]

Key words. Adaptive middleware, error diagnosis, interposition agents, virtual machines.

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.

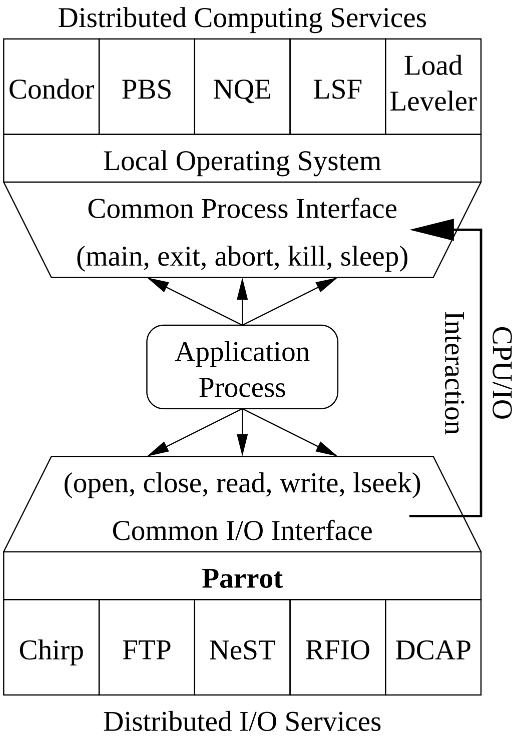


Fig. 1.1. *The Hourglass Model*

The result is an hourglass model of distributed computing, shown in Figure 1.1. 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 [19], 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 impossible 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* [13]. 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.

*Computer Sciences Department, University of Wisconsin

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

	internal techniques				external techniques		
	poly.	**static**	**dyn.**	**binary**	**debug**	**remote**	**kernel**
	exten.	**link**	**link**	**rewrite**	**trap**	**filesys.**	**callout**
scope	library	static	dynamic	dynamic	no setuid	any	any
burden	rewrite	relink	identify	identify	run command	superuser	modify os
layer	fixed	any	any	any	syscall	fs ops only	syscall
init/fini	hard	hard	hard	hard	easy	impossible	easy
aff. linker	no	no	no	no	yes	yes	yes
debug	yes	yes	yes	yes	limited	yes	yes
secure	no	no	no	no	yes	yes	yes
find holes	easy	hard	hard	hard	easy	easy	easy
porting	easy	hard	hard	hard	medium	easy	medium

FIG. 1.2. *Properties of Interposition Techniques*

In this paper, we present practical lessons learned from several years of building and deploying interposition agents within the Condor project. [20, 28, 21, 22] Although the notion of such agents is not unique to Condor [13, 2, 12], 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.

This paper is a condensed version of a workshop paper. Due to space limitations, we have omitted a number of sections and details, indicated by footnotes. The interested reader may find further details in the original paper [23] or in a technical report. [24][2]

2. Interposition Techniques Compared. There are many techniques for interpositioning services between an application and the underlying system. Each has particular strengths and weaknesses. Figure 1.2 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

[2] Omitted: Example applications of interposition agents.

Condor's Java Universe [22] to connect an ordinary InputStream or OutputStream to a secure remote proxy. It is also found in general purpose libraries such as SFIO [25].

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 [20] 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 [8], some implementations of SOCKS [15], as well as our own Bypass [21] 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 [17] toolkit. This technique has been used to "hijack" an unwitting process at runtime [28].

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) [18] 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 [27] 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 [11].

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) 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.

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 1.2 as follows. The polymorphic extension and the remote filesystem are quite easy 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,

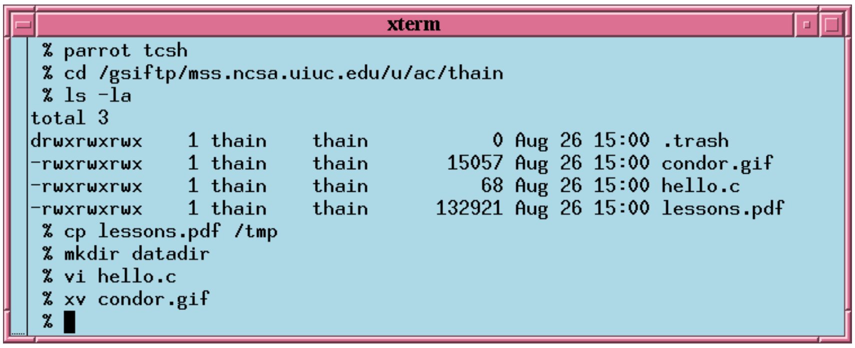
	getpid	**stat**	**open/close**	**read 8KB**	**bandwidth**
unmod	.18±.03 μs	1.85±.09	3.18± .08	3.27± .19	282±13 MB/s
rewrite	.21±.25 μs	1.82±.02	3.21± .05	3.26± .03	280± 7 MB/s
static	.21±.02 μs	1.80±.17	3.59± .05	3.34± .02	280±17 MB/s
dynamic	1.22±.01 μs	3.60±.10	5.53± .06	4.31± .09	278± 4 MB/s
(α unmod)	(6.8x)	(1.9x)	(1.7x)	(1.3x)	(0.99x)
debug	10.06±.21 μs	55.41±.50	42.09± .06	30.99± .26	122± 4 MB/s
(α unmod)	(56x)	(30x)	(13x)	(9x)	(0.43x)

FIG. 2.1. *Overhead of Interposition Techniques*

and binary rewriting—should be viewed as a significant porting challenge that must be revisited at every minor operating system upgrade.

Figure 2.1 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 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.[3]



```
 % 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
 % █
```

FIG. 3.1. *Interactive Browsing with Parrot*

3. Parrot. The Parrot interposition agent attaches standard applications to a variety of distributed I/O systems by way of the debugger trap, described above. Each I/O protocol is presented as a normal filesystem entry under a new top-level directory bearing the name of the protocol. In addition, an optional *mountlist* may be given, which redirects parts of the filesystem namespace to external paths. Figure 3.1 shows Parrot being used with standard tools to manipulate files stored at the Mass Storage Server (MSS) at the National Center for Supercomputing Applications (NCSA) via the Grid Security Infrastructure (GSI) [9] variant of the File Transfer Protocol (FTP).

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 [22]

[3] Omitted: a detailed description of the debugger trap.

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 [6], 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 [8].

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 affected significantly by semantic issues. A summary of the semantics of each of these protocols is given in Figure 3.2.[4]

	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

Fig. 3.2. *Protocol Compatibility with POSIX*

4. 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?[5]

For example, 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. 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, in a manner similar to that of an expert system. Suppose that we attempt to retrieve a file using an FTP GET operation. If the GET should fail, we may hypothesize that the named file is actually a directory. The hypothesis may be tested with a change directory (CWD) command. If that succeeds, the hypothesis is true, and we may return the precise error "not a file." If that fails, we must propose another hypothesis and test it. Parrot performs a number of two- and three-step interviews in response to a variety of FTP errors.

The connection structure of a remote I/O protocol also 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

[4] Omitted: Details of the various protocols supported by Parrot.

[5] Omitted: Several more examples of error transformation.

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.[6]

5. 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.

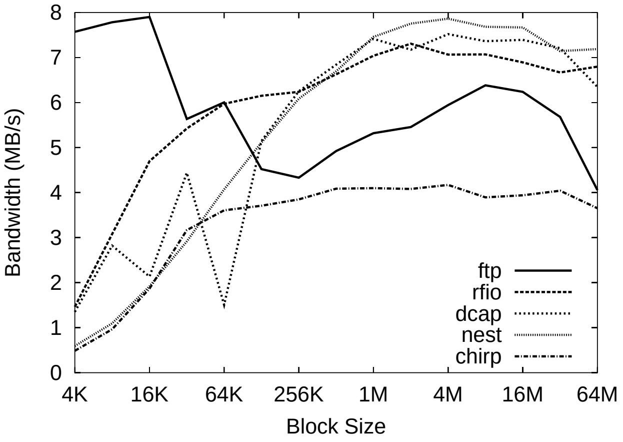


Fig. 5.1. *Throughput of 128 MB File Copy*

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 5.1, 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 all of the protocols must be tuned for optimal performance. 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 or the buffer size to avoid it. This is reasonable advice, given that all of the protocols require tuning of some kind.

Figure 5.2 benchmarks the latency of POSIX-equivalent operations in each I/O protocol. These measurements were obtained in a manner identical to that of Figure 2.1, with the indicated servers residing on the same system as in Figure 5.1. Notice that the latencies are measured in milliseconds, whereas Figure 2.1 gave microseconds.

proto	stat	open/close	read 8KB	write 8KB	bandwidth
chirp	.50± .14 *ms*	.84± .09	2.80± .06	2.23± .04	4.1 *MB/s*
ftp	.87± .09 *ms*	2.82± .26	*(no random access)*		7.9 *MB/s*
nest	2.51± .05 *ms*	2.53± .17	4.48± .14	7.41± .32	7.9 *MB/s*
rfio	13.41± .28 *ms*	23.11± 1.29	3.32± .14	2.85± .18	7.3 *MB/s*
dcap	152.53±16.68 *ms*	159.09±16.68	3.01± 0.62	3.14± .62	7.5 *MB/s*

Fig. 5.2. *Performance of I/O Protocols On a Local-Area Network*

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

[6] Omitted: A discussion of the interface between Parrot and batch systems.

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

Fig. 5.3. *Performance of the Andrew-Like Benchmark*

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 benefitted 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. [11] 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 the **delete** stage, the tree is deleted.

Figure 5.3 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.

6. 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. 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 [26] 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 [14, 7], 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 [16] describes how the Berkeley sockets interface is surprisingly hard to multiplex. T. Garfinkel [10] describes the subtle semantic problems of sandboxing untrusted applications.

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

7. Acknowledgments. We thank John Bent and Sander Klous for their help 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, (1998), pp. 207–233.

[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, 2000, pp. 161–163.

[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] 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.

[7] D. Cheriton, *UIO: A uniform I/O system interface for distributed systems*, ACM Transactions on Computer Systems, 5 (1987), pp. 12–46.

[8] 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.

[9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A security architecture for computational grids*, in Proceedings of the 5th ACM Conference on Computer and Communications Security Conference, 1998, pp. 83–92.

[10] 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.

[11] 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 (1988), pp. 51–81.

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

[13] M. Jones, *Interposition agents: Transparently interposing user code at the system interface*, in Proceedings of the 14th ACM Symposium on Operating Systems Principles, 1993.

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

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

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

[17] 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 (1995), pp. 37–46.

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

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

[20] M. Solomon and M. Litzkow, *Supporting checkpointing and process migration outside the Unix kernel*, in Proceedings of the USENIX Winter Technical Conference, 1992.

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

[22] ———, *Error scope on a computational grid*, in Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing, July 2002.

[23] ———, *Parrot: Transparent user-level middleware for data-intensive computing*, in Proceedings of the Workshop on Adaptive Grid Middleware, September 2003.

[24] ———, *Parrot: Transparent user-level middleware for data-intensive computing*, Tech. Report 1493, Computer Sciences Department, University of Wisconsin, December 2003.

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

[26] 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.

[27] 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.

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

Edited by: Wilson Rivera, Jaime Seguel.
Received: July 14, 2003.
Accepted: September 1, 2003.