Deploying Complex Applications in Unfriendly Distributed Systems with Parrot ((PREPRINT VERSION))

Douglas Thain (thain@cs.wisc.edu)
Computer Sciences Department, University of Wisconsin, United States

Sander Klous (sander@nikhef.nl)
National Institute for Nuclear and High Energy Physics, Netherlands

Miron Livny (miron@cs.wisc.edu)
Computer Sciences Department, University of Wisconsin, United States

Abstract. Ordinary applications struggle to benefit from distributed computing. Most applications are designed for the safe confines of a single workstation and are not prepared to encounter the new interfaces and failures that are endemic to distributed systems. To solve this problem, we present Parrot, an interposition agent that connects standard, unmodified applications to distributed systems. Parrot makes use of the debugging interface to trap and modify an application's system calls. This interface is heavyweight but foolproof: Parrot can operate on any program, script, or multi-process conglomerate. We explore how Parrot can be used to attach a variety of remote I/O protocols and we explain why a new protocol, Chirp, is necessary to support real applications. We present a case study of SP5, a high-energy physics application that requires distributed computing to achieve its production goals. Using Parrot, we successfully deploy SP5 into a fault-prone distributed system.

Keywords: Distributed computing; interposition agents; fault tolerance.

1. Introduction

Ordinary computing applications struggle to benefit from the promises of distributed computing. Although there exist countless systems for harnessing remote processors and accessing remote data, many place stringent requirements on the applications that they accept. A batch system might require that all programs be a single executable performing no interprocess communication. A distributed file system may provide unusual consistency semantics that are at odds with a user's expectations. Many experimental systems expect users to re-write their software to take advantage of new features, while many production systems expect users to have administrator privileges on all machines on a network.

These restrictions are unacceptable in the real world. Typical developers write their applications on standalone machines, making liberal use of complex and powerful libraries and systems. By re-using existing tools, developers are able to concentrate on their craft rather than reinventing computing from the ground up. Software is created, debugged, and validated on ordinary workstations long before any thought turns to distributed computing. Users in a corporate or academic environment are not likely to have administrator privileges on a large number of machines.

Regardless, a wide variety of human endeavors hope to benefit from large scale distributed computing. Physical science has an unlimited appetite for simulation capacity: researchers in astronomy, chemistry, and physics can explore a larger parameter space or reduce error bounds simply by harnessing more cycles. Similar needs may be found to the fields of video production, data mining, and finance, to name a few. How are we to make distributed computing accessible to ordinary programs?

Parrot is our answer to this challenge.

Parrot is an *interposition agent*, a piece of software that inserts itself between a ordinary program and the operating system. When used in an unfriendly distributed system, Parrot provides the illusion of a user's home environment, including files user identities, and more. Parrot can customize an application's environment to create a synthetic namespace formed from multiple remote services. In addition, Parrot is able to hide network outages, server crashes, and other failures that are endemic to distributed systems.

Although the notion of interposition agents is not new (15, 1, 14), they have seen relatively little use in production systems. This is due to a variety of technical and semantic difficulties that arise in connecting real systems together. For example, many different I/O protocols may be attached to an application, but few provide the full range of POSIX semantics expected by many applications. For this reason, we have created our own protocol, Chirp, which provides the precise semantics that applications expect.

Parrot is an extension to an existing operating system; it augments file-handling capabilities without affecting a process' ability to interact with other processes on the same machine or over a network. Parrot is considerably simpler than other tools like virtual operating systems such as UML (6) and virtual machines such as VMWare [1], both which require the user to build and maintain virtual networks, large filesystem images, and all the elements of an isolated operating system in miniature. Parrot consists of a single executable measuring only 8.4 MB with all options enabled, and as small as 1 MB in minimal configuration.

[1] `http://www.vmware.com`

As a motivating example, we describe how Parrot can be used to deploy an application into a real distributed system. This application, SP5, is representative of the applications described above; it consists of multiple processes and complex libraries that defy most distributed computing and file systems. In particular, we explore the problem of working through an aggressive firewall that discards active TCP connections.

This paper is a modified version of a workshop paper (21) also available as a technical report. (22) The description and evaluation of SP5 is entirely new material. The comparison of I/O protocols has been re-written, but the data is taken from the earlier paper. Due to space limitations, a comparison of several interposition techniques has been removed, and some details of performance have been omitted.

2. Example Application: SP5

SP5 [2] is a software component of the BaBar high-energy physics experiment in progress at the Stanford Linear Accelerator Center. A large amount of computation is needed to understand the response of the BaBar experimental apparatus. The physics interactions are mimicked by the simulation of random particle collisions known as events. These events are fed to a simulation of the detector geometry, which results in a trace of all of its output signals. These traces are fed into a reconstructor that infers the nature of the original collision events. SP5 is the first phase of this computing activity, which is known as *monte carlo production.*

At an abstract level, SP5 operates by first loading data that describes the configuration of the detector and the physics of particle generation. Once loaded, it enters a compute-intensive phase where it generates an arbitrary number of events that can each be summarized in 10-100 kilobytes. As is common for many batch computing applications, more computation directly results in a higher quality result. As the measurements of the BaBar experiment are progressing, the accuracy of the results improves and the interest shifts toward rare phenomena. To compare the results with models, the accuracy of the simulation needs to improve as well, which means a higher number of events needs to be simulated and more resources are needed.

The computing needs of the international BaBar collaboration exceed the resources available at any one of its constituent research labs and universities. However, all of them put together should provide sufficient computing hardware. The problem of distributing this application

[2] `http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Production`

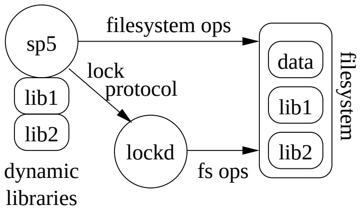

Figure 1. The Structure of SP5 and its Data

is one of organization: the more efficiently the hardware can be used, the higher quality the end result will be. However, the aggregate system must also be simple enough that resources do not sit idle while humans struggle to configure and debug computers. Such a large-scale computing environment that crosses administrative boundaries is sometimes known as a *computational grid*.

In theory, SP5 has the right structure for distributed computing. The initial data can simply be distributed to a number of processors, production can be performed in parallel, and the produced events can be returned to a central site. Once initialized, any processor can produce an arbitrary number of events, so the number of processors can be chosen to balance startup time against desired throughput.

In practice, SP5 has a number of complexities that make it difficult to deploy in a distributed system. Figure 1 shows some of these complexities. A standard filesystem contains the SP5 executable and scripts, several dynamic libraries, the input configuration, and the output events. The program is wrapped by a script that establishes environmental settings and verifies the integrity of the file system before invoking the program. It also makes use of several dynamically-loaded libraries, particularly the Objectivity [3] database, which manages the configuration and event data structures.

Objectivity is a decentralized, cooperative database built on top of a standard filesystem. Consistency management, access control, and crash recovery are performed cooperatively by clients rather than enforced by a server. A minimal central server assists only with a locking protocol. To read the configuration data or write events, the client library requests a lock from the lock server, manipulates the file system directly, and then releases the lock.

This structure is quite reasonable when viewed alone, but is difficult to adapt to an existing distributed system. For example, the filesystem activity of the Objectivity client library cannot be carried over a standard distributed filesystem. The delayed-writeback semantics of NFS (18) clients are too weak for database structures, while the strict

[3] `http://www.objectivity.com`

open-close semantics of AFS (13) would result in data loss on the append-only transaction log. Objectivity does have the capability to speak NFS directly to a server, bypassing the buffer cache, but deploying this requires superuser privileges at both the client and the server; an unlikely capability in a grid computing environment.

It might be argued that SP5 ought to be restructured in order to take better advantage of distributed computing. For example, if it was collected into a single, statically linked executable, it could take advantage of Condor's (19) checkpointing and migration features. If re-written to a parallel computing interface such as MPI (7), it might be more easily parallelized. However technically tempting such options might be, they come at an enormous cost in development and debugging labor. (In fact, developers have recently performed a large amount of work on SP5; the next release will incorporate features that ease distributed computing.) If a way is found to deploy applications on a distributed system without making modifications, then both hardware and human resources can be used more efficiently.

Further, a computational grid is inherently an unfriendly environment with its own challenges. Installing most software on a new cluster is a labor-intensive process that defies automation: executables, scripts, and libraries must be unpacked and installed; environment variables and other settings must be configured; database structures must be initialized; dependent software must be discovered and installed. Some software expects a uniform user database across multiple machines; this is an impossibility on a computational grid. The nature of a distributed environment ensures that network outages and performance variations are common events.

This is the challenge of distributed computing in the real world: For sound social and technical reasons, we may not modify either applications or the underlying computing systems. To accomplish real work, we must find a way to transparently connect complex applications to unreliable systems. For this, we use interposition agents.

3. Interposition Agents

An *interposition agent* (agent for short) is a piece of software that inserts itself between two existing layers of software in order to modify their discourse. By inserting an agent rather than modifying an existing piece of software, we may measure, debug, and enhance an application without requiring intimate knowledge of its innards. An interposition agent has many uses in a distributed system:

Seamless integration. The most common use of an interposition agent is to connect an application to a new resource, such as a storage device, without requiring any special changes or coding in the application. For example, Parrot allows an application to seamlessly connect to a remote storage server. The application merely perceives it to be an ordinary file system.

Improved reliability. In general, remote data services are far less reliable than local filesystems. Remote services are prone to failed networks power outages, expired credentials, and many other problems. An interposition agent can attach an application to a service with improved reliability. For example, Rocks (23) 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. An interposition agent can be used to create a private namespace for each instance of an application, thus allowing many to run simultaneously while keeping their I/O activities separate. For example, several 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.

Remote dynamic linking. Although dynamic linking offers many technical advantages for programs that share code or data, it presents a number of social problems. It is all too easy to migrate an application only to discover that needed libraries are missing, or worse yet, that the available libraries are the wrong version. An interposition agent can solve these problems by allowing an application to link against libraries stored at a single, well-known server.

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 agent to an application, a user may easily generate a trace or summary of I/O behavior and observe precisely what the application does.(20).

4. Parrot

Parrot is an interposition agent that provides the features discussed above for standard Unix applications. It observes and potentially mod-

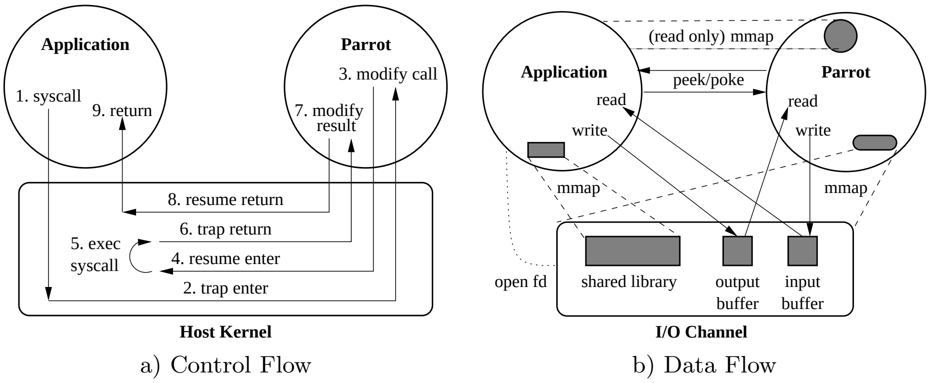

Figure 2. Interpositioning via the Debugger Interface

ifies the interaction between an unmodified process and the operating system kernel using the standard debugging interface. Alexandrov et al. (1) have shown how the Solaris *proc* interface may be used to instrument a process. 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 2.a shows the control flow necessary to trap a system call through the *ptrace* interface. Parrot registers its interest in an application process with the operating system kernel. At each attempt by the application to invoke a system call, the host kernel notifies Parrot. Parrot may then modify the application's address space or registers, including the system call and its arguments. Once satisfied, Parrot instructs the host kernel to resume the system call. At completion, Parrot is given another opportunity to make changes before passing control back to the kernel and the application.

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 Parrot. 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 2.b 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 file operations, or can be mapped into the address space of Parrot. Although this provides high-bandwidth read access, writing to this file is not permitted. [4] Instead, 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 into assisting Parrot. This is accomplished by converting many system calls to *preads* and *pwrites* on a shared buffer called the *I/O channel*. This is an ordinary file created by Parrot, passed implicitly, and shared among all of its children. Parrot maps the I/O channel into memory, to minimize copying, while all of the application processes simply maintain a file descriptor pointing to the I/O channel.

For example, suppose that the application issues a *read* on a remote file. Upon trapping the system call entry, Parrot examines the parameters of *read* and retrieves the needed data. These are copied directly into a buffer in the I/O 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 I/O channel, unaware of the activity necessary to place it there.

This method differs significantly from that demonstrated by UFO.(1) The UFO method only traps calls to *open* and immediately fetches the whole file in question so that later operations may access it at full speed locally. In contrast, Parrot traps all file operations. This permits more fine-grained control and semantic power, but comes at a cost in performance.

Figure 3 measures this overhead. 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.

[4] 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.

	getpid	open/close	read 1B	read 8KB	bandwidth
unmod	.18±.03	3.18± .08	.93± .23	3.27±.19 μs	282±13 MB/s
parrot	10.06±.21	42.09± .06	19.38±1.03	30.99±.26 μs	122± 4 MB/s
change	(56x)	(13x)	(21x)	(9x)	(0.43x)

Figure 3. Overhead of Interposition with Parrot

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 *parrot* case shows the same program with Parrot attached.

Parrot has a significant performance overhead: most system calls are an order of magnitude slower. More importantly, bandwidth in and out of the process is reduced by half, due to the extra data copy incurred by the techniques described above. There do exist other interposition techniques with lower overhead, but good performance comes by sacrificing reliability. A complete discussion of this problem is given in our earlier paper. (21) That said, this level of overhead is acceptable for applications such as SP5 that have both CPU and I/O intensive phases.

5. Protocols and Semantics

The primary use of Parrot is to attach applications to remote storage devices. For example, Figure 4 shows Parrot used to attach a variety of standard system utilities to a secure FTP server.

To this end, Parrot has a variety of drivers for various I/O protocols. As mentioned, the File Transfer Protocol (FTP) and its secure GSI (2) variant are supported. The Chirp protocol was designed by the authors to provide remote I/O with semantics very similar to POSIX. A standalone Chirp server is distributed with Parrot. The NeST protocol is the native language of the NeST storage appliance (4), 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 by the high-energy physics community to provide access to hierarchical mass storage devices such as Castor (3) and DCache (8).

Each type of remote storage device is presented to the user as a filesystem entry naming the protocol and server name. For example, a Chirp server named *bird.cs.wisc.edu* is made available under the path */chirp/bird.cs.wisc.edu*. Access to local files, such as */etc/passwd* is

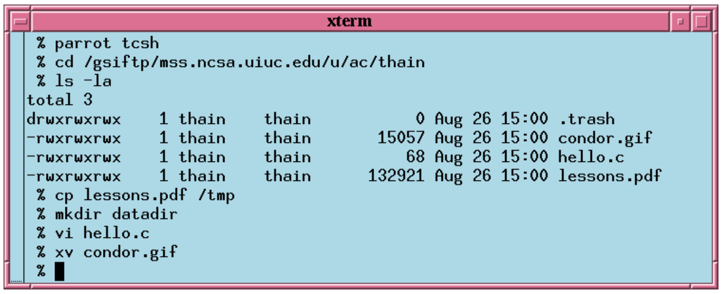

Figure 4. Interactive Browsing with the Parrot Interposition Agent

unchanged. A user may also specify a *mountlist* that maps logical path names to other physical devices. This allows Parrot to create a custom namespace for a program, perhaps even emulating the environment of another machine. For example, this mountlist maps */mydata* to an FTP server and the standard library directory to a Chirp server:

```
/mydata   /ftp/ftp.cs.wisc.edu/datadir
/usr/lib  /chirp/bird.cs.wisc.edu/usr/lib
```

Not all of the protocols supported by Parrot are equal. 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. A summary of the semantics of each of these protocols is given in Figure 5.

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.

With the exception of FTP, all of the protocols provide inexpensive random (i.e. out-of-order) 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. Such behavior is needed for SP5, which manipulates small portions of large database files. The sequential nature of FTP requires that Parrot read and write entire files as they are opened and closed.

Directories are supported completely by Chirp, NeST, and RFIO; one may create, delete and list their contents. DCAP does not cur-

protocol	model	discipline	dirs	stat	links	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 5. Protocol Compatibility with POSIX

rently support directory access, although this may be added in a later version. [5] 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-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. In addition, the *stat* interface allows the operating system to hint at an ideal block size for file I/O. This interface allows Parrot to tune many applications to use a large block size and hide the increased latency of interposition.

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. Even 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.

[5] DCAP is typically used in conjunction with a kernel NFS client that provides access to metadata and directories.

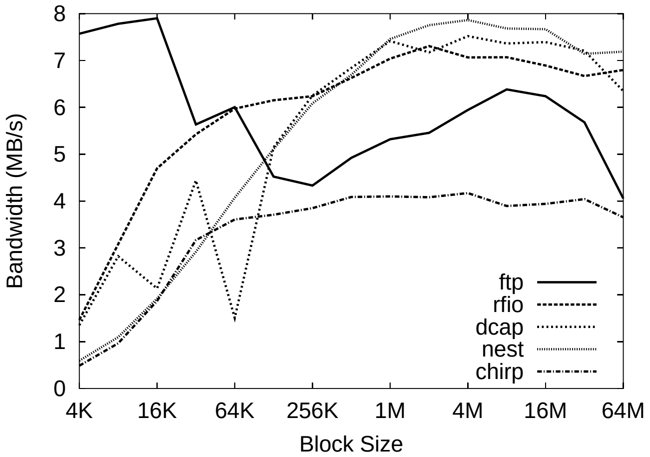

Figure 6. Throughput of 128 MB File Copy

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 6, 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 *stat* block size hint.

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 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.

Figure 7 measures the latency of POSIX-equivalent operations in each I/O protocol when carried over a 100 Mb ethernet. These measurements were taken in an identical manner to those in Figure 3. Notice that the units have increased from microseconds to milliseconds.

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

proto	**stat**	**open/close**	**read 1B**	**read 8KB**
chirp	.50± .14	.84± .09	.61± .04	2.80± .06 *ms*
ftp	.87± .09	2.82± .26	*(no random access)*	
nest	2.51± .05	2.53± .17	2.96± .17	4.48± .14 *ms*
rfio	13.41± .28	23.11± 1.29	.50± .06	3.32± .14 *ms*
dcap	152.53±16.68	159.09±16.68	40.05±0.17	3.01±0.62 *ms*

Figure 7. Performance of I/O Protocols On a Local-Area Network

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 optimization, 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 ptrace interposition by several orders of magnitude.

We conclude with a macro-benchmark similar to the Andrew (13) benchmark. This benchmark emulates the file-system intensive activity of a program developer using 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. Finally, the tree is deleted.

Figure 8 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. Naturally, data movement runs at

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

Figure 8. Performance of the Andrew-Like Benchmark

network speeds, but 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 many random accesses 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 with caching enabled. 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.

Although Parrot is capable of using a variety of storage services, we conclude that Chirp, by virtue of its low latency and similarity to POSIX, is a good choice for general-purpose I/O workloads.

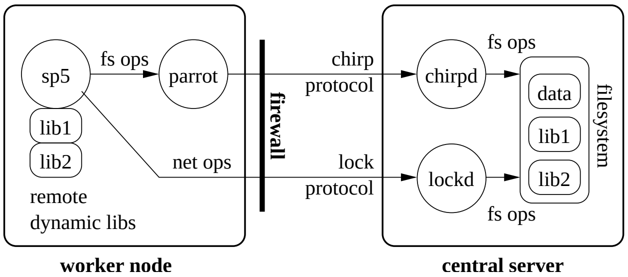

Figure 9. Deploying SP5 and Parrot in a Distributed System

6. Deploying SP5 with Parrot

With this knowledge in hand, we are prepared to use Parrot to deploy SP5 into a distributed system. Figure 9 shows how the pieces fit together. The configuration data and output events are stored in an Objectivity-managed filesystem on a well-known *central server*. A central lock server process assists with mutual exclusion. A number of *worker nodes* are used to execute instances of SP5. Access to a number of worker nodes at various institutions is obtained by way of Condor-G (10) and the Globus toolkit (9). No special software is installed on any of the worker nodes, nor do we have superuser access in these environments, so we rely on Parrot to carry all of the SP5 filesystem operations back to a Chirp server deployed at the central server. SP5 is run by Parrot with the following simple mountlist:

```
/     /chirp/central-server.nikhef.nl/
```

This mountlist is comparable to an NFS client that mounts its root filesystem from a remote device: all executables, dynamic libraries, and other program components are loaded from the central server via the Chirp protocol. Parrot makes local copies of executables; this is a technical necessity, because Unix can only execute a program identified by a local file name. All data files are accessed remotely without caching so as to avoid introducing a consistency problem into the database.

In addition to the filesystem, a number of other small settings were necessary to fully emulate the home environment. For example, the Objectivity libraries examine the POSIX user identifier and host name in order to implement access control on the database. Because worker machines may not necessarily share a user database with the central server, we instruct Parrot to trap these system calls and change the results to match what would be seen at the central server.

Aggressive firewalls posed a serious problem to the deployment of this system. It is quite common for a computing cluster to be connected to the public Internet by way of a firewall and network address translator (NAT). In the clusters targeted by this application, the NAT permits cluster nodes to initiate outgoing TCP connections to the public Internet, but prohibits incoming connections, except as necessary to dispatch batch jobs. To translate external addresses into internal addresses, the NAT must keep state about every TCP connections that it carries.

The problem arises when a NAT must discard TCP connections that it perceives to be idle. Each connection consumes some state in the firewall, so it cannot keep them forever. The most aggressive NAT that we have encountered discards TCP connections that have been idle for only one minute. When this happens, there is a double penalty: not only is the connection lost, but the NAT does not even return an RST packet indicating that the connection was lost. The result is that both sides think the connection is present but lossy, and retry up to their maximum timeouts, which can range from minutes to hours.

This problem was deadly to SP5. Once it initialized, the lock server connection was held open and idle, while the Chirp connection was only used for the output of each event, at intervals of slightly more than a minute. While SP5 was processing the first event, the NAT would discard the TCP connections. A short time later, the entire system would hang while attempting to write out the first event.

Although we would like to simply discount this firewall as an aberrant device, we cannot consider reconfiguring its timeout to be a reasonable solution. For the same reasons that we cannot install software or act as the superuser on a worker node, we cannot expect to reconfigure the network interior at will. (In fact, we later discovered that these are the factory settings for the NAT in question. We do not relish the idea of negotiating with a network administrator every time this model of NAT is encountered.)

One stop-gap solution to this problem is to change the network endpoints to generate enough traffic to keep the NAT state alive. For example, we may modify the networking stack at the central server to send TCP keepalives at the rate of several per minute. We applied this technique in order to preserve the connection between SP5 and the lock server. However, it is unsatisfying because it requires administrator privileges on at least one end.

A more comprehensive solution is to make the network protocol recoverable, so that the failure of the TCP connection becomes a harmless event. For example, we modified Parrot so that a failed Chirp connection was recovered by reconnecting and reopening the needed files. With this recovery method, we could afford to make the Chirp connection *fail-fast* (12); hence, any delay of greater than thirty seconds was assumed to be a transient network failure and would result in

distance	method	proto	cpu	time to init	time per event
same	os	files	1 GHz	446 ± 46 s	64 s
same	parrot	files	1 GHz	668 ± 26 s	65 s
same	parrot	chirp	1 GHz	777 ± 48 s	66 s
lan	os	nfs	1 GHz	4464 ± 172 s	113 s
lan	parrot	chirp	1 GHz	4505 ± 155 s	113 s
wan	parrot	chirp	2.5 GHz	6275 ± 330 s	88 s

Figure 10. Performance of SP5 and Parrot Deployed in a Distributed System

disconnection and recovery. This solution is more robust than simply applying keepalives – it also tolerates the crash and recovery of the Chirp server – but could only be implemented because we are free to modify the Chirp protocol.

The Chirp recovery method reveals an old problem in the design of distributed filesystems. Strict POSIX semantics require that an application holds references directly to files rather than names. That is, once an application opens a file by name, it keeps access to that file even if the name is deleted or renamed. Distributed file systems such as NFS (18) and AFS (13) solve this problem by exposing inode numbers to clients. When recovering from a disconnection, NFS and AFS clients can be assured of access to the correct files by referring to the inode numbers. Chirp cannot do this directly; the Chirp server is implemented on top of an ordinary file system and thus can only open files by name. However, the Chirp protocol can verify that the binding between names and inodes has not changed after a recovery by simply querying inode numbers with the *stat* operation. If they have not changed, recovery is successful. Otherwise, recovery has failed, Parrot forces the application to fail immediately, and the batch system becomes responsible for re-starting it from the beginning.

After addressing the problem of recovery, we turned to issues of performance. Figure 10 shows the run-times of SP5, gradually increasing the logical and physical distance between it and its data on the central server. As discussed earlier, SP5 begins with an I/O-intensive startup phase, and then settles into a CPU-intensive phase of configurable length. As the distance increases, the I/O-intensive phase pays a increasing price, but the CPU-intensive phase is relatively stable.

The first line of Figure 10 shows the performance of unmodified SP5, as in Figure 1. The application is run in “validation mode”, producing additional histograms to cross check the results. Furthermore, it produces a full debugging output so that we can verify that it ran correctly.

As a result the production is approximately a factor of 5 slower than the standard production on this machine. The average and standard deviation (σ_n) of initialization times are shown along with the average time to process an event. Each measurement of the initialization time is the result of 10 trials. The time to process one event is an avarage of 2000 events. A small numbers of outliers beyond 5 σ_n were attributed to unrelated network traffic and discarded. Each succesive line adds one component in order to measure its contribution. The final line measures the complete system as depicted in Figure 9.

The first row measures the baseline performance of SP5, unmodified, run on the same machine as its data. It initializes in 443 seconds and then processes one event every 64 seconds. The second row adds Parrot, but without any remote I/O or other features; SP5 just accesses local files through Parrot. The third row adds Chirp, but without a network; SP5 accesses a Chirp server on the same machine using Parrot. As can be seen, both Parrot and Chirp slow down initialization, but have little effect on event processing.

The fourth and fifth rows show the performance of SP5 accessing its data over a local area network (latency $130 \pm 10\mu s$.) In the fourth row, SP5 is using a kernel-level NFS client to access Objectivity's files, ignoring potential consistency problems due to caching. In the fifth, SP5 is using Parrot and Chirp to accomplish the same task safely without a cache. Although initialization is an order of magnitude slower than the unmodified case, we can see that the performance of Chirp is comparable to NFS. The overhead is more a function of the network than of Parrot or Chirp.

The final row shows the performance of SP5 accessing its data over a wide-area network (latency $654 \pm 50\mu s$) via the firewall discussed above. Notice that the performance numbers are not directly comparable, as the CPU is about 2.5 times as fast as the others. (The CPUs in a real-world distributed system are rarely identical.) However, we may see the same qualitative result as the other lines: initialization is slow, but event processing is reasonable.

Overall, the BaBar experiment must process billions of events to complete the necessary simulation. In the worst case of accessing data over a wide area network, the cost of computing events equals the cost of initialization at only 70 events. Given that a typical instance of SP5 processes 10,000, we conclude that the cost of remote execution, while significant, can be amortized across a large run.

7. Conclusions

Interposition agents bridge the gap between applications and systems when neither are available for modification. By raising the level of abstraction on which an application executes in a batch system, we are able to provide a transparent and reliable environment, even in an uncertain environment. We have shown that deploying a complex application into a distributed system is quite feasible with the help of Parrot. This initial study into distributing SP5 is not complete; we still need to better understand the scalability of SP5 and Chirp.

As our discussion of remote I/O protocols illustrates, virtualizing an existing interface is a subtle problem that has implications for reliability, recoverability, and performance. The low-level details of a protocol – for example, how many network connections it consumes – can determine whether it usable by ordinary applications. Although interface virtualization is a common technique (16, 5), the problems 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 (17) describes how the Berkeley sockets interface is surprisingly hard to multiplex. T. Garfinkel (11) describes the subtle semantic problems of sandboxing untrusted applications.

For more information: `http://www.cs.wisc.edu/condor/parrot`

Acknowledgements

We thank Concezio Bozzi and the BaBar monte carlo production team for their assistance with SP5 and the production site.

References

1. Alexandrov, A., M. Ibel, K. Schauser, and C. Scheiman: 1998, 'UFO: A personal global file system based on user-level extensions to the operating system'. *ACM Transactions on Computer Systems* pp. 207–233.
2. Allcock, W., A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke: 2000, 'Protocols and Services for Distributed Data-Intensive Science'. In: *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*. pp. 161–163.
3. Barring, O., J. Baud, and J. Durand: 2000, 'CASTOR Project Status'. In: *Proceedings of Computing in High Energy Physics*. Padua, Italy.
4. Bent, J., V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny: 2002, 'Flexibility, Manageability, and Performance in a Grid Storage Appliance'. In: *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*. Edinburgh, Scotland.

5. Cheriton, D.: 1987, 'UIO: A Uniform I/O system interface for distributed systems'. *ACM Transactions on Computer Systems* **5**(1), 12–46.
6. Dike, J.: 2000, 'A user-mode port of the Linux kernel'. In: *Proceedings of the USENIX Annual Linux Showcase and Conference*. Atlanta, GA.
7. Dongarra, J. J. and D. W. Walker: 1996, 'MPI: A Standard Message Passing Interface'. *Supercomputer* pp. 56–68.
8. Ernst, M., P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman: 2001, 'dCache, a Distributed Storage Data Caching System'. In: *Proceedings of Computing in High Energy Physics*. Beijing, China.
9. Foster, I. and C. Kesselman: 1997, 'Globus: A metacomputing intrastructure toolkit'. *International Journal of Supercomputer Applications* **11**(2), 115–128.
10. Frey, J., T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke: 2001, 'Condor-G: A Computation Management Agent for Multi-Institutional Grids'. In: *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*. San Francisco, California, pp. 7–9.
11. Garfinkel, T.: 2003, 'Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools'. In: *Proceedings of the Network and Distributed Systems Security Symposium*.
12. Gray, J.: 1986, 'Why do Computers Stop, and What Can Be Done About It?'. In: *Proceedings of the Symposium on Reliable Distributed Systems*. pp. 3–12.
13. Howard, J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West: 1988, 'Scale and Performance in a Distributed File System'. *ACM Transactions on Computer Systems* **6**(1), 51–81.
14. Hunt, G. and D. Brubacher: 1999, 'Detours: Binary Interception of Win32 Functions'. Technical Report MSR-TR-98-33, Microsoft Research.
15. Jones, M.: 1993, 'Interposition Agents: Transparently Interposing user Code at the System Interface'. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. pp. 80–93.
16. Kleiman, S.: 1986, 'Vnodes: An architecture for Multiple File System Types in Sun Unix'. In: *Proceedings of the USENIX Technical Conference*. pp. 151–163.
17. Metz, C.: 2002, 'Protocol Independence Using the Sockets API'. In: *Procedings of the USENIX Technical Conference*.
18. Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon: 1985, 'Design and implementation of the Sun network filesystem'. In: *Proceedings of the USENIX Summer Technical Conference*. pp. 119–130.
19. Solomon, M. and M. Litzkow: 1992, 'Supporting Checkpointing and Process Migration Outside the Unix Kernel'. In: *Proceedings of the USENIX Winter Technical Conference*. pp. 283–290.
20. Thain, D., J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny: 2003, 'Pipeline and Batch Sharing in Grid Workloads'. In: *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*. Seattle, WA.
21. Thain, D. and M. Livny: 2003a, 'Parrot: Transparent User-Level Middleware for Data-Intensive Computing'. In: *Proceedings of the Workshop on Adaptive Grid Middleware*.
22. Thain, D. and M. Livny: 2003b, 'Parrot: Transparent User-Level Middleware for Data-Intensive Computing'. Technical Report 1493, University of Wisconsin, Computer Sciences Department.
23. Zandy, V. and B. Miller: 2002, 'Reliable Network Connections'. In: *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking*. Atlanta, GA, pp. 95–106.