TRANSPARENTLY DISTRIBUTING CDF SOFTWARE WITH PARROT

Douglas Thain and Christopher Moretti, University of Notre Dame, Notre Dame, IN 46556, USA
Igor Sfiligoi, INFN-Frascati and Fermi National Laboratory, Batavia, IL 60510, USA

Abstract

The CDF software, like many toolkits for high energy physics, was designed to be executed in a dedicated environment, with easy access to a large set of executables, shared libraries, and configuration files. In order to meet the computing needs of CDF, it is necessary to move the software on to a computational grid. However, in such an environment, the necessary software components are not easily accessible, nor is it practical to copy the software in its entirety to each CPU. To address this problem, we are applying the Parrot virtual filesystem to the CDF software stack. Parrot allows an application to access remote data sources as if they were local filesystems. No special privileges or application changes are needed to employ Parrot, so it is well suited to the environment of a computational grid. We describe the strengths and weaknesses of this approach and measure the performance of CDF code in the wide area. Although Parrot imposes a high cost on individual system calls, the overhead is approximately five percent of runtime when amortized across CPU-intensive jobs.

INTRODUCTION

The CDF software, like many toolkits for high energy physics, was designed to be executed in a dedicated environment, with easy access to a large set of executables, shared libraries, and configuration files. As might be expected, the CDF code is a conglomeration of contributions from many different authors working in different languages and contexts. Each component of the system has many runtime dependencies that are difficult to decompose.

In order to meet the computing needs of CDF, it is necessary to move the software on to systems such as EGEE and the Open Science Grid and take advantage of as many CPUs as possible. [11] However, in a grid computing environment, the necessary software components are not easily accessible. We cannot expect every grid administrator to install a distributed shared filesystem and mount a server for the benefit of any one application. On the other hand, it is not practical to copy the CDF software to each node of the grid as is needed: the total system consists of GB of data in many thousands of files. Transferring this amount of data per job will not scale, nor can we expect users to disentagle the various components merely to run codes on the grid.

For the sake of end users, we would like to provide a filesystem interface to CDF code, no matter where jobs happen to execute. The challenge is to do this without requiring special privileges. We have developed a prototype solution to this problem by applying the Parrot [15] virtual filesystem to the CDF software stack. Parrot allows an application to access remote data sources as if they were local filesystems. No special privileges or application changes are needed to employ Parrot, so it is well suited to the environment of a computational grid. We describe the strengths and weaknesses of this approach and measure the performance on CDF simulation codes in the wide area. Although Parrot imposes a high cost on individual system calls, the overhead is approximately five percent of runtime when amortized across a CPU-intensive application.

OVERVIEW OF PARROT

Parrot is a personal virtual filesystem for performing Unix-like I/O on remote data services, including HTTP, FTP, GridFTP [2], Nest [5], Chirp [14], RFIO [3], SRB [4], and DCAP [6]. Parrot presents these services to the application as entries in the filesystem namespace. For example, a user may start a shell by invoking `parrot tcsh` and then simply access files in `/http/www.fnal.gov/index.html` using normal command-line tools like `cp`, `ls`, and `vi`.

A custom namespace may be constructed for applications running under Parrot. The user may define a *mountlist* which is similar in spirit to the system-wide `fstab` file in Unix. For example, the following mountlist allows a CDF user to employ a directory on a web server at Fermilab as a consistent global home directory:

```
/home/cdfsoft =
          /httpfs/cdfsoft.fnal.gov/base
```

Parrot operates by running applications using the `ptrace` debugging facility in the operating system kernel. As shown in Figure 1, each time the application attempts a system call, the application is halted, and Parrot is notified by the kernel. Parrot then interprets the arguments to the system call, and then implements the system call, perhaps by invoking operations on a remote storage device.

This mechanism can be used to run most end-user applications. Programs need not be modified, re-compiled, or re-linked to run under Parrot. The `ptrace` mechanism allows Parrot to trace multiple processes at once, so applications may be complex scripts or interpreted programs. Currently, Parrot works with a wide variety of programs, including most standard system tools, and many applications written in C, C++, FORTRAN, Java, Perl, Python,

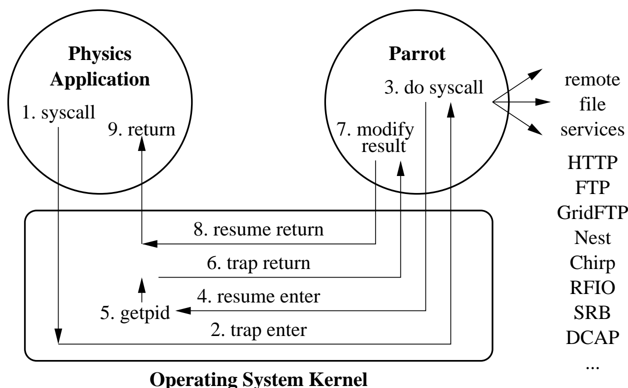

Figure 1: Trapping System Calls in Parrot
Parrot traps system calls via the `ptrace` *interface. (1) The application attempts a system call, which the kernel (2) forwards to Parrot. (3) Parrot implements the system call perhaps by contacting a remote file service. (4) Internally the system call is converted (5,6) to a harmless* `getpid`*. (7) Parrot modifies the system call result and passes control back to the kernel (8) and then to the application (9).*

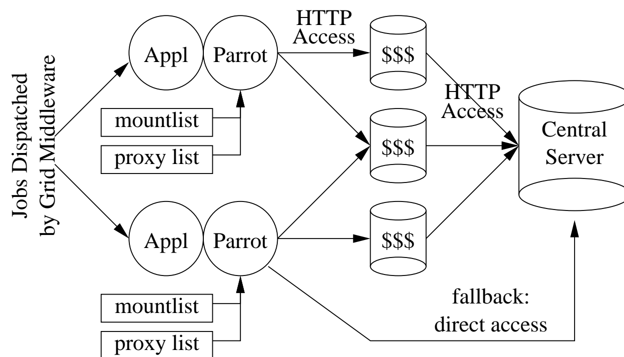

Figure 2: HTTP as a Filesystem
Jobs are dispatched to the grid middleware, augmented with Parrot. Once running, the applications access data remotely via the HTTP protocol. A mountlist controls the visible namespace. Intermediate caching proxy servers are selected by Parrot. In the event of a proxy failure, the home server is accessed directly.

and shell scripting languages. There are some restrictions. The `ptrace` interface will not allow Parrot to run `setuid` applications. More importantly, Parrot must have detailed knowledge of the underlying system calls, so the current implementation is closely tied to the Linux kernel.

Parrot is designed to work best with the Chirp [14] I/O protocol. A Chirp server distributed with Parrot provides an I/O interface very similar to that of Unix, along with a fine-grained security mechanism. However, deploying a new type of server into a production system is a complex matter. Administrators may be (rightfully) suspicious of a new server, which may have bugs introducing security concerns. Likewise, protocols using custom port numbers present difficulty in traversing firewall and network translation devices.

Instead, we have chosen to use HTTP as the data service for distributing CDF code to applications running on a grid. Users may employ servers already established by institutions, traverse most firewalls designed to accomodate the protocol, and employ shared proxy servers in order to share bandwidth and storage space. The CDF software may be modified at the central repository, but it is read-only with respect to clients on the grid.

HTTP AS A FILESYSTEM

Although HTTP has many administrative benefits, it has one severe drawback: it is not a true filesystem protocol. HTTP allows a client to get and put whole files over the network, it is missing three key components necessary to support arbitrary filesystem actions: directory listings, metadata operations, and partial-file access. Even the simplest of applications and scripts require these operations (metadata retrieval is the most common filesystem operation [7]), so it is not practical to ask users to use HTTP "carefully" and avoid certain operations.

We note that *some* HTTP servers may support *some* of these operations. For example, many HTTP servers return an HTML formatted directory listing when a web browser requests a path that corresponds to a directory. A client could attempt to parse this output to obtain a structured directory listing. In response to a carefully formatted HEAD request, some HTTP servers may respond with the size, owner, and other metadata describing a file. HTTP 1.1 defines range units that allow a client to request part of a file, although the server is not obliged to respect the request. If we carefully prescribe what HTTP server, version, and options to employ, it would be possible to use HTTP directly as a filesystem. However, we wish to make our solution as generally applicable as possible; many people make use of an institutional server that they cannot modify. Therefore, we decline these approaches to using HTTP.

Instead, we use an approach that relies only on the most basic facilities found in any HTTP server. On top of HTTP, we layer filesystem functionality and call the combination HTTP-FS: HTTP with FileSystem extensions. First, we ask the user to run a small script `make_httpfs` on the data to be exported. This script recursively scans the directories and in each creates a file `.httpfsdir`. This file contains a listing of the directory and the metadata for each entry. Using this, the three problems are solved as follows:

- **Directories.** When accessing a pathname `/x/y`, Parrot first attempts to access `/x/y/.httpfsdir`. If successful, the path is known to be a directory. On failure, Parrot then accesses `/x/y` directly. Note that the procedure cannot be reversed, because if `/x/y` is a directory, many servers will generate a "helpful" HTML directory listing that would appear to be an ordinary file. The basic type of a file is determined only by the accessibility of these path names.

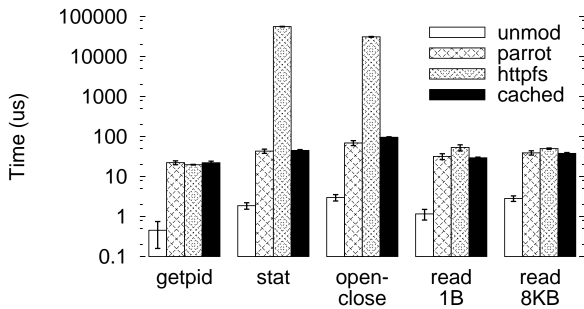

Figure 3: System Call Latency

The impact of Parrot and remote file access on system call latency. Parrot itself increases the latency of system calls by an order of magnitude compared to an unmodified program. HTTP adds another order of magnitude, but this can be amortized with caching. Note the overall effect on applications in Figure 5 is much less dramatic.

- **Metadata.** When obtaining metadata about a file (such as the information returned by `stat`), Parrot obtains the `.httpfsdir` file in the directory containing that file. This file must be searched for the entry corresponding to the name, and then the appropriate metadata returned to the application.

- **Caching.** To support efficient partial file access, Parrot maintains an HTTP cache on disk. When a file is first opened, it is retrieved in its entirety, and then accessed on local disk. This cache also applies to the `.httpfsdir` auxiliary files. Even with cached data, the process of determining the type and metadata of a pathname may involve several expensive disk operations, so these are cached in the internal memory of Parrot as well.

Figure 2 shows how all of the pieces fit together. Simple jobs are submitted to the grid middleware. These jobs are merely scripts which obtain Parrot via an HTTP download, and then use HTTP via Parrot to load scripts and run the desired executable. For each job, a custom mountlist and list of proxy servers is given to Parrot to create the desired environment. Parrot selects an intermediate proxy server at random, so that multiple jobs may share access to nearby data. If access to a proxy should fail, direct access to the central HTTP server is used as a backup.

The entire system relies only on the presence of the `wget` tool at each grid node. Using only this tool, Parrot itself can be retrieved and then used to access any scripts or executables necessary to run the desired job.

PERFORMANCE

We may consider the performance of this system from two perspectives. First, what is the cost of the low-level interposition technique that traps and transforms system calls? Second, what is the effect of this low-level cost on real applications?

Throughput	Traps/s	Unmodified	w/Parrot
1 MB/s	15k	199s	199s
8 MB/s	120k	199s	199s
20 MB/s	300k	201s	205s
40 MB/s	600k	204s	215s

Figure 4: Bandwidth Available to Parrot

The impact of Parrot on available I/O bandwidth. This chart shows a synthetic code consuming CPU time while attempting to load data continuously from local disk at the specified rate. There is no overhead for speeds up to 8MB/s, which are rarely exceeded by real codes.

Appl	Unmod	w/Parrot	w/HTTPFS
Real Simulation	11.9h	12.1h	12.5h
Toy Monte Carlo	16.6m	16.8m	17.3m

Figure 5: Performance of CDF Applications

The impact of Parrot and HTTP-FS on overall execution time of simulation codes. Although the cost of individual system calls is increased, the actual effect on real CPU-bound physics jobs is approx five percent. This overhead is acceptable, given that it allows the harnessing of CPUs that would otherwise go idle.

Figure 3 shows the low-level expense of trapping system calls via `ptrace` and forwarding them to remote I/O services. This measurement was performed on dual 2.8GHz Pentium IV CPUs running Linux 2.4.21 and Parrot 2.0.13. Measurements were performed by issuing 1000 cycles of 1000 iterations of system calls measured with the system timer. Four configurations are shown: unmodified calls applied to a local filesystem, calls trapped by Parrot but applied to a local filesystem, and calls trapped by Parrot and applied to HTTP-FS, with and without caching.

As can be seen, the addition of Parrot without even adding remote I/O increases system call latency by an order of magnitude. The cost of a getpid() rises from less than 1 μs to over 10 μs. HTTP-FS itself also adds several more orders of magnitude: each `stat` and `open` results in many round trips to establish a TCP connection and make several HTTP transactions. However, once a file is opened, data is streamed to the application, so reads have a low latency. This cost can be amortized by adding the data and metadata caching described above. With caching enabled, the cost of HTTP-FS is minimal.

Figure 4 shows the overall effect of trapping system calls via `ptrace` and forwarding them to local services for CPU intensive jobs. Each line shows the runtime of a CPU-intensive job accessing local storage at a target rate. This test clearly shows that this in kind of application, the overhead introduced by Parrot is minimal. This is important, since one does not want to penalize local access just to gain access to remote files.

Figure 5 shows the overall effect on CDF simulation codes. Both of these applications were run unmodified on a local disk, with Parrot on a local disk, and then with Parrot accessing data via HTTP-FS over the wide area. As can be seen, the increased expense of system calls and data transfer only increases runtimes by about five percent. Thus, if this technique allows us to harness many more CPUs than would otherwise be available, the overall system throughput will increase.

IMPLEMENTATION STATUS

For most of our tests, the system used was composed of a few well controlled worker nodes and using direct HTTP connections to the Web server. Using HTTP caching and running over wide area network was tested, but we didn't go beyond a proof of principle.

The reason for such limited set of tests was due to lack of time. Several of the features we needed were developped for CDF just very recently and it took some time to polish out the initial bugs. In the near future, we plan to test Parrot with many more real user jobs on a much wider scale, using real Grid worker nodes and making full use of HTTP caching. We will report on this in future publications.

RELATED WORK

Parrot is first introduced in [15] and placed in the larger context of *tactical storage* in [14]. It has also been used to implement remote access in the context of the BaBar experiment [8]. The notion of trapping system calls to implement remote access was first demonstrated by the Remote UNIX [10] system of Litzkow and Solomon, which later became part of Condor [9]. Remote UNIX relies on recompiling target applications; whereas Parrot operates on arbitrary, unmodified programs. The technique of trapping system calls via the debugging interface was pioneered by Alexandrov et al [1], who demonstrated access to remote HTTP services, but did not address the problems of directories or fault tolerance. The private namespace feature is inspired by the user-level mount capability found in the Plan-9 [12] operating system. Other storage systems that provide some form of user-level interposition include dCache [6] and SRB [4], both which provide approaches based on dynamic linking.

CONCLUSION

As J. Schopf [13] has observed, the primary obstacle to large scale grid computing is not performance, but the usability of complex systems. Distributed systems by their very nature are less usable, less reliable, and less predictable than centralized systems. In this work, our aim is to make grid computing systems as easy to use as local systems without a large performance sacrifice.

REFERENCES

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

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

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

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

[5] 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 *Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[6] 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.

[7] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed fi le system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

[8] S. Klous, J. Frey, S.-C. Son, D. Thain, A. Roy, M. Livny, and J. van den Brand. Transparent access to grid resources for user software. *Concurrency and Computation: Practice and Experience*, to appear.

[9] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Eighth International Conference of Distributed Computing Systems*, June 1988.

[10] M. J. Litzkow. Remote Unix - Turning idle workstations into cycle servers. In *USENIX Summer Technical Conference*, pages 381–384, 1987.

[11] M. Neubauer, S. Sarkar, I. Sfi ligoi, F. Wuerthwein, M. Norman, S.-C. Hsu, and E. Lipeles. OSG-CAF - a single point of submission for CDF to the Open Science Grid. In *Computing in High Energy Physics*, February 2006.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[13] J. Schopf. State of grid users: 25 conversations with UK eScience groups. Argonne National Laboratory Tech Report ANL/MCS-TM/278, 2003.

[14] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *International Conference for High Performance Computing, Networking, and Storage (Supercomputing)*, November 2005.

[15] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.