CDF software distribution on the Grid using Parrot

G Compostella[1]**, S Pagan Griso**[2]**, D Lucchesi**[2]**, I Sfiligoi**[3] **and D Thain**[4]

[1] INFN-CNAF, viale Berti Pichat 6/2, 40127 Bologna, ITALY
[2] Dipartimento di Fisica G. Galilei, via Marzolo 8, 35131 Padova, ITALY
[3] Fermi National Accelerator Laboratory, Batavia, IL 60510, USA
[4] University of Notre Dame, Notre Dame, IN 46556, USA

E-mail: `compostella@pd.infn.it, simone.pagan@pd.infn.it`

Abstract. Large international collaborations that use decentralized computing models are becoming a custom rather than an exception in High Energy Physics. A good computing model for such big collaborations has to deal with the distribution of the experiment-specific software around the world.
When the CDF experiment developed its software infrastructure, most computing was done on dedicated clusters. As a result, libraries, configuration files and large executables were deployed over a shared file system. In order to adapt its computing model to the Grid, CDF decided to distribute its software to all European Grid sites using Parrot, a user-level application capable of attaching existing programs to remote I/O systems through the filesystem interface.
This choice allows CDF to use just one centralized source of code and a scalable set of caches all around Europe to efficiently distribute its code and requires almost no interaction with the existing Grid middleware or with local system administrators. This system has been in production at CDF in Europe since almost two years.
Here, we present CDF implementation of Parrot and some comments on its performances.

1. Introduction

The CDF software, like many other software packages for high energy physics, is a set of executables, scripts, shared libraries and specific configuration files whose usage has been approved by the collaboration for the production of high quality physics results.
Some components are customized for CDF, while others taken from standard physics code libraries, but more generally all the different pieces of software are highly interconnected, can come from several contributions written by different authors and in multiple programming languages, and may have many runtime dependencies very difficult to decompose.

For any given simulation or analysis job, it is very difficult to determine a priori what set of files are needed, since different software modules may be accessed according to the configuration files used or to the kind of job under execution. That's why since its first development and during its many evolutions over time, CDF software was always supposed and designed to be executed in a dedicated environment.

In order to cope with its increasing computing power needs, and to take advantage of as many CPUs as possible, in the past few years CDF started to move a relevant part of its offline computing from the usage of dedicated farms towards Grid environments such as LCG and OSG. As a result of this effort, 3 new portals for job submission are now available to CDF users: each of these portals sends users jobs to European, American and Asian Grid sites respectively in a way that is completely transparent to the CDF collaborator; CDF users can indeed continue to submit jobs as usual, as if the accessed resources were local, since all the complexity of the Grid job submission is hidden by a set of custom middleware software systems.

However, when dealing with a heterogeneous environment as the Grid, many problems have to be taken into account; in the following we will focus on the issue that the necessary CDF software components cannot be easily accessible in the variety of sites accessed by our Virtual Organization.

First of all we cannot expect every Grid site administrator to install a distributed shared filesystem and mount a server for the benefit of one single VO. Additionally, the software is constantly under development and debugging: although some of its releases are frozen since a long time, users may want to access its nightly build release to take advantage of new avaible tools, debugged applications or experimental features not included in previous releases. Keeping all the software releases up to date to the version developed at Fermilab would require a supplementary maintainance effort for the site admins.

On the other hand, copying the whole CDF software to each worker node would pose severe scalability issues, since the total system consists of several GBs of data spread in many thousands of files. Moreover, one of the main concerns behind the transition of CDF offline computing to the Grid world was to have a minimal impact on the users, hiding from them any kind of complication related to the new tools under usage. That's why we don't expect users to take care of disentagling the various components of the software accessed by their jobs just to be able to run their codes on the Grid.

Our aim is then to provide a filesystem interface to CDF code, accessible no matter where jobs happen to execute, easy to mantain and to update, transparent to the users and requiring no changes to their existing codes.

In the following section, we will discuss the requirements that should be met by our filesystem interface. We will then move to the description of Parrot, an interposition agent which allows to attach existing programs to remote I/O systems through the filesystem interface. Then we will describe our implementation of the Global Read Only Web File System (GROW-FS) and its implication for the consistency and data integrity of the distributed CDF software. Finally, we will briefly review the performances of our solution and its impact on typical CDF jobs.

2. Design Requirements

Our challenge is to be able to distribute experiment specific software to Grid sites without the need of changing our computing model and without asking users to modify their approach to job submission; this will allow to have a minimal impact on the operation duties of our analysis facilities administrators, while letting the physicists focus on the quality of their results instead of the details of the underlying computing system. In particular, we are interested in developing a filesystem interface matching the following constraints:

- Transparent file access. As already discussed, we don't want users to spend time redesigning their applications just to be able to run them on the Grid. Access to the software must be transparent to them and everything should work as if the filesystem used was local.
- No need for special privileges. Jobs on the Grid cannot be expected to run with any kind of privilege allowing them to load kernel modules, install device drivers or mount custom filesystems.
- No need to deploy new services on the Grid site. Usually each LCG or OSG site supports many different VOs, we cannot expect site administrators to setup and mantain new special services just to support a single VO in the list, if any other existing service can be used adequately.

- Limited network connectivity. Usually computing sites do not guarantee unlimited network access. Some sites have security policies that limit network traffic to specific activities, preventing the use of filesystem technologies that employ other ports or protocols. Other sites might only permit outgoing TCP connections from the worker nodes to the internet, preventing the use of filesystem technologies that rely on the use of incoming connections to the clients.
- Data integrity. The main interest of the collaboration is to ensure the correctness of scientific outcomes: any published official result must be computed using a standard and approved software version, whose code is shared among all collaborators. So we can disregard the importance of privacy of the filesystem and focus on offering a method of verifying the integrity of the code in use.
- Scalability. Usually jobs are submitted in large batches that may share a large fraction of their libraries and configuration files. To be scalable, the system must allow the usage of shared local caches available to multiple jobs running in the same site.
- Limited number of data repositories. Our solution must be easy to mantain and shouldn't require any extra effort by the collaboration in terms of manpower, so it must be able to function properly without the need of deploying many code repositories.

Considering the existing filesystems, it's hard to find any that satisfies all these constraints. For example NFS requires administrator privileges to be installed at the client side and has poor wide area performance; AFS has better wide area performance, but still requires administrator intervention. Staging the whole software package via GridFTP or HTTPS would offer a suitable security model, but would also require to copy several GBs of data to every worker node in the system; since encryption is used during data transfers, they could not rely on shared proxy caches. Plain HTTP could take advantage of caching, but would not provide data integrity. Moreover none of HTTP, HTTPS, or GridFTP alone would provide transparent file access.

Our idea is to start from a well known, widely used protocol like HTTP and add to it the features needed in order to satisfy all our design constraints.
This choice is based on the assumption that every Grid site will have firewall and security settings allowing outbound HTTP connections from its worker nodes, so that our solution will work no matter where the jobs happen to execute. However, as this protocol by itself cannot create a filesystem, we will combine it with several additional software components, creating a new system called GROW-FS, as we describe in the following section.

3. Architecture of the GROW-FS Filesystem
CDF jobs may run on different computing centers that can be separated by wide area networks. To build our system we create a job wrapper that executes users code under the supervision of Parrot, an interposition agent that transparently connects the jobs to the filesystem. Each site can have a proxy server that caches the files downloaded from a central code repository where the entire CDF software distribution is stored. We use the standard HTTP protocol for data transfer, assuming that all sites will allow their nodes to perform outbound HTTP connections. Figure 1 shows an overview of the system architecture.

3.1. Parrot
Parrot [1, 2] is an open source software that acts as an interposition agent; it can be used to attach existing applications to filesystems without requiring code or kernel changes. Parrot intercepts the system calls of an application through the `ptrace` debugging interface: system calls that do not affect I/O are passed through unmodified, while those affecting I/O like `open` and `read` are executed by Parrot on behalf of the application. This allows the interposition

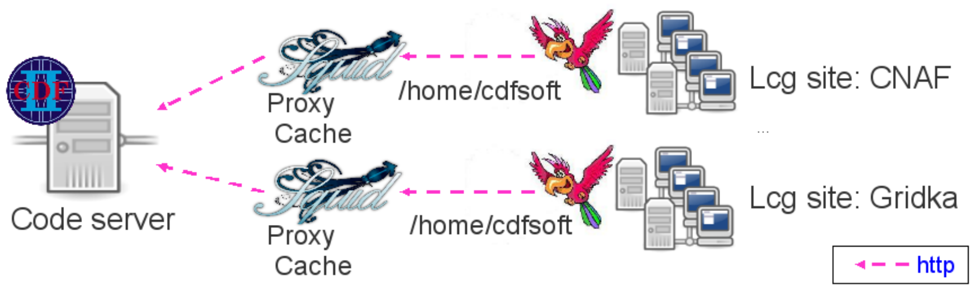

Figure 1. System architecture. CDF jobs can be sent to various Grid sites, each job runs under the supervision of Parrot, that intercepts its system calls to the directory `/home/cdfsoft`. HTTP connections and proxy servers are used to transfer files from a central code server to the worker node. In the end, jobs run as if executing locally available programs.

agent to contact a remote storage resource, get the necessary files, and then pass them to the application simulating the access to a local file.

Parrot can use various software drivers to communicate with different storage devices such as HTTP, FTP, GridFTP, RFIO, and others. Each service is presented to the application as a top level directory entry. For instance, a user may start a shell by invoking `parrot bash` and then simply access files in `/http/www.fnal.gov/index.html` using command-line tools like cp, ls, and vi.

Additionally, custom namespaces can be defined for applications running under Parrot. The user can define a *mountlist* which is similar to the system-wide `fstab` file in Unix. For example, by defining the following mountlist a CDF user can employ a directory on a web server located Fermilab as a consistent global home directory:

```
/home/cdfsoft = /growfs/cdfsoft.fnal.gov/software
```

These settings will configure the Parrot agent to translate all the system calls made by an application running under its supervision to the directory `/home/cdfsoft` into file transfers from `cdfsoft.fnal.gov/software` performed using the HTTP protocol.

In CDF setup, Parrot uses also a local cache on the worker node to store the files downloaded from the code server, and is configured to use a proxy server by choosing the closest to the Grid site from a static list of available machines.

This mechanism can be used to run most end-user applications, without modifying or recompiling them. Using `ptrace`, Parrot is able to trace multiple processes concurrently, so applications may be complex scripts or interpreted programs. Currently, Parrot works with a wide variety of programs, that range from most standard system tools to user applications written in many different languages.
However, since Parrot must have detailed knowledge of the underlying system calls, there are some restrictions that make the current implementation closely tied to the Linux kernel.

In conclusion, Parrot is an excellent tool to attach existing applications to remote filesystems without any change to their code. In the following, we describe our choice for the remote filesystem used for distributing CDF code to applications running on the Grid.

3.2. Bulding a filesystem from HTTP

The choice of HTTP as the data transfer protocol to serve CDF code to applications will allow users to traverse most firewalls designed to allow the protocol and employ proxy servers in order to share bandwidth and storage space among their jobs.

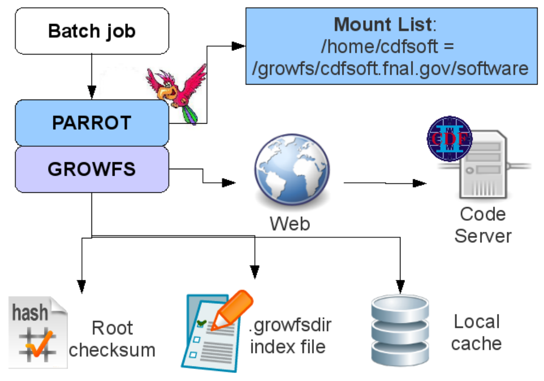

Figure 2. Each user job runs under the supervision of Parrot, that traps the programs system calls via the `ptrace` interface, and executes them on behalf of the application. When using GROW-FS, an index file is used to keep track of directory structure and integrity of data, while individual files are stored in a local cache.

This will also allow to store, mantain and update the necessary CDF software components at a central repository, represented by a simple web server, allowing clients on the Grid to have read-only access to the code.

Unfortunately HTTP alone is just a data transfer protocol that allows a user to access the content of a file whose name is already known. Moreover, it is not a complete filesystem protocol by itself, since in general it doesn't offer the features necessary to support arbitrary filesystem actions, like directory listings, metadata operations, and partial-file access. Although some HTTP servers may support some of these operations, it is important to note that they are not required by the protocol, and have a strong dependence on the configuration of the server.

We want our solution to be independent from the type, version and configuration of the web server used, so we will follow an approach that relies only on the capability to serve files to clients, building on top of HTTP a layer with filesystem functionalities.

First we ask the software repository manager to run a script that scans the directory to be exported and creates a single index file called `.growfsdir` that contains a listing of every file together with its metadata (owner, size, access permissions, etc.) and a checksum of the file.

This index file is downloaded by Parrot upon first reference to a GROW-FS filesystem and is then used to perform every operation. Upon loading the index file, the entire directory structure of the exported code and the corresponding HTTP address for each file is known and available to the application. Thanks to the cached directory metadata, Parrot can tell if a file is up-to-date without any network communication, and can avoid spending any network activity on file lookups that do not exist. The additional information on the SHA1 checksum of each file referenced by the `.growfsdir` index can be used to ensure integrity of the transfered data. Figure 2 shows an overview of the Parrot+GROW-FS system in a single job.

3.2.1. Data Integrity Due to the importance of the reliability of the physics output results obtained by CDF jobs, one of the problems we want to address is the possible corruption of data transfered from the code server. To ensure and verify the integrity of data in the filesystem, we rely on the checksum of each file. The index file itself is checksummed to produce a root checksum, which is downloaded on Parrot startup together with the `.growfsdir` and used to check the integrity of the index file. Additionally, as each requested file is loaded into the worker node local cache, its checksum is computed and compared to the checksum in the index file.

3.2.2. Consistency Requirements As the software hosted in the code server is subject to frequent updates and bugfixes, consistency management problems may arise. When new versions of the software are published or modified, the software manager will modify the code repository accordingly and regenerate the index file. Then the new software must be propagated to the execution sites: because components of one software version might not be compatible with another, immediate propagation of the updates to running jobs should be avoided.

To implement these requirements the root checksum is used as an indicator of a data change, therefore is never cached, but downloaded fresh at the beginning of each new job execution. If the cached index file does not match the root checksum, a new copy is downloaded. Similarly, any data file not matching the checksum in the index file is fetched again.

It may happen that a job starts running and the software manager updates the filesystem and rebuilds the index file before the job completes. If every file needed by the job has already been loaded into the local cache, then the program won't see any change in the home filesystem, and will complete using the old version of the software. If the job needs a file from the code server which was not changed during the update, it will load that file and see no other changes. Finally, if the job needs a file that is not cached and has changed, then GROW-FS will observe an inconsistency between the checksum of the fetched file and the one stored in the index file. When an inconsistency is detected, first of all the file in question is redownloaded along with a flag that forces the proxy server to flush and refetch its copy as well. This handles the possibility of data corruption while transfering the file and ensures the coherence of the cache used.

If the inconsistency remains, this means that the file hosted in the code server is newer with respect to the copy of the index file cached on the worker node. The root checksum is then refetched from the code server. If the root checksum has not changed, then it is assumed that the code server is in the process of being updated. In that case, the index file is downloaded again after an exponentially increasing delay.

If it has changed, it means that the software update is complete; in that case there is no possibility of completing the program using the old version of the software. If this happens, Parrot should download the new index file and restart the user job from scratch using the new software version along with the updated root checksum. At the moment, this feature is still under development, since it raises questions still under investigation about how to deal with the intermediate outputs of the job.

This procedure will ensure the consistency of the software used on job execution and the coherence of the cache used by Parrot.

4. Usage experience

The system described is in production since almost two years in the CDF job submission portal accessing European resources of the LCG Grid.

A single code server hosted at CNAF-Tier1 is used and kept up to date with respect to software releases developed at Fermilab. This code server hosts many of the different official frozen releases of the CDF software together with its nightly development build, amounting to almost 20 GB of data.

Typically the creation of the `.growfsdir` index file takes around 30 minutes on our Intel(R) Xeon(TM) CPU 3.06GHz with 4 GB of RAM running Scientific Linux CERN Release 3.0.6 (SL). The resulting index file weights around 20 MB uncompressed. This ensures that the startup cost added to every user job in terms of execution times due to the download of the index file is kept low, considering that usual CDF analysis and simulation jobs rely on user sandboxes of the order of few hundreds of MBs.

Multiple SQUID proxy servers have been deployed close to the biggest sites in Europe used by CDF for its offline computing: local proxy caches dedicated to CDF are available at CNAF-Tier1, GridKA, and IN2P3. The job wrapper used to invoke Parrot and run the user code is able to choose dinamically the proxy closest to the execution site without any user intervention. Since many jobs are submitted in batches that share the same code base, this lowers dramatically the load on the main code server in terms of HTTP requests to serve.

Taking into account the performances of the system, the first thing to check is the cost of the low-level interposition technique that traps and transforms system calls forwarding them to the remote I/O service. There is a small effect introduced by Parrot which increases system calls latency by an order of magnitude, as described in [3].

However, the overall impact of the introduction of Parrot in a typical CDF job can be considered negligible: thanks to the usage of a local cache on the worker node, Parrot downloads the files needed by the application only once; additionally, by exploiting the described interposition mechanism, Parrot downloads just what's needed by the application and nothing else. The overhead introduced by Parrot using GROW-FS is found to increase runtimes of typical CDF simulation jobs by about five percent. Considering that those jobs have typical completion times of about 20 hours, the impact on users is kept low.

Moreover, since this technique allows us to use many more CPUs than would otherwise be available, the overall system throughput increases despite the longer runtimes, thus making the cost to benefit ratio of this solution very favourable for the CDF collaboration.

5. Conclusions

As CDF started to move its offline computing to the Grid to exploit more CPU power, the main constraint behind the transition was to be able to avoid changing its computing model, that was designed with dedicated resources in mind.
Being the Grid a complex distributed system, the primary obstacle to the transition was to keep the same usability CDF collaborators were used to have on the original systems.

Focusing on the problem of distributing the experiment specific software in this heterogeneous environment, we found that the usage of Parrot with GROW-FS smoothly integrates with our grid-oriented computing tools, is easy to deploy and mantain, requires no extra effort by CDF personel, and proves to be very effective in hiding the complexity of the distributed systems from our users. This allows us to make grid computing for CDF collaborators as easy to use as local systems without any noticeable performance sacrifice.

6. Related Work

Parrot was first introduced in [1] and then placed in the larger context of tactical storage in [4]. The notion of trapping system calls to implement remote access was first demonstrated by the Remote UNIX [5] system, which later became part of Condor [6]. The private namespace feature is inspired by the user-level mount capability found in the Plan-9 operating system [7].
An earlier version of this work based on a different remote filesystem setup is described in [3].

References

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

[2] Parrot official Web site: `http://www.cse.nd.edu/~ccl/software/parrot/`

[3] Thain D, Moretti C and Sfiligoi I 2006 Transparently Distributing CDF Software with Parrot *Proceedings of International Conference on Computing in High Energy Physics (CHEP06)*

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

[5] Litzkow M J 1987 Remote Unix - Turning idle workstations into cycle servers *USENIX Summer Technical Conference*

[6] Litzkow M J, Livny M and Mutka M 1988 Condor - a hunter of idle workstations *Eighth International Conference of Distributed Computing Systems*

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