SCRATCHFS: A FILE SYSTEM TO MANAGE SCRATCH DISK SPACE FOR GRID JOBS

Leandro Franco * and Fabio Hernandez †
IN2P3/CNRS Computing Centre
Lyon, France

Abstract

Managing the temporal disk space used by jobs in a farm can be an operational issue. Efforts have been put on controlling this space by the batch scheduler to make sure the job will use at most the requested amount of space, and that this space is cleaned up after the end of the job. ScratchFS is a virtual file system that addresses this problem at the file system level for grid jobs as well as conventional jobs. Designed to be on top of a real file system, the virtualization layer is responsible for controlling the access to the underlying file system. For conventional (i.e. local) jobs, the access control to the disk space is based on standard Unix file permissions. In the case of grid jobs, ScratchFS uses only the grid credentials of the job to grant or deny access to the scratch disk space in a fully transparent way from the application point of view. Therefore, several grid jobs submitted by different individuals can be executed in the same worker node, even if their grid identities are all mapped to the same local user identifier, while preserving the confidentiality of each job scratch space. Designed to be extensible, it can be configured to collect information about the file system usage and to enforce quotas, among other things. In addition, a simplified interface is provided to ease the integration with a batch scheduler. In this paper we present how this virtualization layer is used by our local batch scheduler and we explain the implementation of the system in detail, along with some comments on its security and benchmark results compared to a real file system.

INTRODUCTION

The static or dynamic association of grid and local (to a particular resource or site) identities may not be appropriate from the operations point of view. Those associations are needed for actually running processes or accessing data belonging to a particular individual (identified by his grid credentials) in a particular machine.

In the static association approach, a mapping between grid credentials and existing Unix accounts is specified. The administrator of a particular resource may map all the individuals of a virtual organization to the same account or perform a one-to-one mapping if confidentially of the data is necessary.

In the dynamic approach, a Unix account is dynamically selected (when the mapping is necessary) and the grid credential is mapped to it. A set of pooled accounts is devoted to this by the site administrator. If a dynamically selected one-to-one mapping is desired, a big enough set of accounts must be created in advance for this purpose.

The main motivation for using different accounts for mapping different individuals is to preserve confidentiality in the data created by the jobs run on behalf of them.

We present in this work a mechanism to control access to file systems based on grid credentials. Our main target is to provide a more appropriate tool for managing the scratch disk space local to a worker node where grid jobs are executed.

RELATED WORK

Extensive work has been done in the area of sand boxing [2][3], two of the most known representatives are probably *Janus* [5] and *Subterfugue* [6], which try to control untrusted programs by intercepting system calls and granting or denying access to resources according to a predefined policy. The tools in this category have been designed to monitor programs in the usual Unix-like world and the extension to the grid would not be straightforward. On the other hand, as a framework, Subterfugue can be more easily adapted but unfortunately it can run up to ten times slower than the original process, which can be a price too high for some production centres. Another interesting idea is *SBLSM: a sandbox mechanism for Linux* [4] where instead of being intercepted in user space (like Subterfugue or the others applications using process tracing), the system calls are tracked in kernel space using the Linux Security Modules (LSM) framework. This idea is more general and could potentially allow a security control of all the resources in a machine, but we have to wait for the LSM project to achieve a more widely distribution and utilization.

From the side of the grid world, similar ideas are being considered, in particular, the idea of *Virtual Environments* [7]. They can be seen as an abstract notion to encapsulate a grid job, different implementations can be handled with the notion of factories. As an example of such implementations we have dynamic Unix accounts, sandboxes and virtual machines.

For the moment, we think that having virtual machines as virtual environments can be too expensive in performance terms and it also provides a bigger functionality than needed. Sandboxes are lighter since they usually intercept the communication between jobs and resources instead of creating a *virtual* space for the task, but as mentioned before, no propositions have been found for a grid world.

* leandro.franco@cc.in2p3.fr
† fabio@in2p3.fr

Finally, having dynamic accounts is an acceptable solution and is what is mostly used but the transformation from grid identities to local users has potential problems, for instance, we can not guarantee that the function mapping grid users to local users is injective, which can lead to two different *real* individuals sharing the same account (this is an important issue when using account pools).

PROPOSED SOLUTION

Our proposed solution is in its simplest way an implementation of a virtual environment reduced to file access. In this case, we could say that it is a *Sand Box File System*, where particular properties can be checked while running jobs.

Controlling access and checking space usage are the most important features of the system. Access control to files is based on grid identities (usually certificate-based) and not on the Unix accounts to which they are mapped. In addition, the system can be configured to also control access based on normal Unix accounts, mainly for making the integration of the system with existing infrastructures.

To implement the virtual file system we used FUSE[8], which provides the mechanisms in kernel space to intercept and replace the system calls related to Input/Output, allowing us to execute our own versions of such calls from a daemon running in user space. The basic diagram showing us where ScratchFS is located with respect to the kernel and the user space can be seen in figure 1.

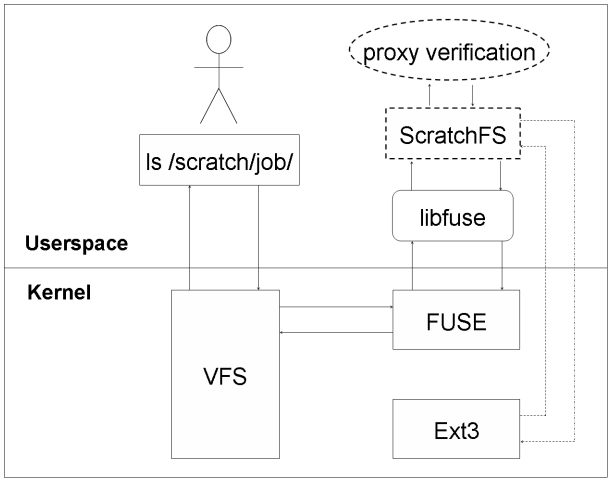

Figure 1: Basic diagram of the ScratchFS integration

MAIN FEATURES

Enforcing quotas

The virtualization layer can be configured to control the usage of disk space for a particular job. When a job tries to exceed its allocated disk quota, the virtualization layer will signal it by returning the appropriate return code through the system call in a completely transparent way to the application. From the operations point of view, this allows a better control on how the disk resource is used by the job and prevents that a malfunctioning job can badly impact other jobs running in the same worker node. In addition, the quota for any virtual file system managed by ScratchFS can be modified at any time in the file system life, within the limits of the physical capacity of the underlying real file system.

GSI Authentication

The most interesting feature from the Grid point of view is the access control based on the Grid Security Infrastructure (GSI) [1] [9]. Since the grid jobs usually have an identity of its owner, the system controls access to files according to this identity instead of the local Unix account the grid identity is mapped to. This has the benefit that several grid jobs belonging to different individuals can be executed by the same Unix account in the same worker node, while preserving the confidentiality of the data created by them in the local file system used for temporary storage.

To achieve this we borrowed an idea from the Andrew File System (AFS) world, where a Process Authentication Group (PAG) is assigned to the job's processes, and this PAG is then associated to the GSI Proxy in the file system. This is similar to the relationship between PAGs and tokens made in AFS, but ours is a simplified version running in user space. Using the PAGs we can guarantee that all the descendant processes will inherit the proxy, which means that all the processes in the same process group will have access to the file system.

Controlling Access for Local Users

The system also allows access control to files based on Unix accounts. A list of the accounts allowed to access the virtual file system can be specified at creation time and all of them will be considered as *owners* of the file system. The list of authorized accounts is associated to the virtual file system and every operation checks the requester against the specified list. This is a general approach and a more fine grained method can be implemented, like an Access Control List (ACL)-based mechanism.

PRINCIPLE OF OPERATION

ScratchFS has been designed to be easily integrated to the local resource management system, which is in charge of allocating and controlling all the resources for a job. In particular, it is responsible for creating the virtual file system for the job, launching the job's processes and destroying the virtual file system at the end of the job. Below we present some considerations for those steps.

Creating and mounting the file system

When preparing the environment for executing a job, the batch system will interact with ScractchFS by using one of the provided APIs. A distinction must be made between the Unix account used for creating and mounting the virtual file system and the one actually executing the job. Once this space is created it will be set as the current working directory of the job.

Running the job

Although it is presented as one call in the library, this process is a little more complicate than the last one. The *run* function will set the PAG of the process and will execute it, for that, we need to run the call as the *Scratch Executer* .

The steps that it will follow are :

- Ask for the PAG or set it up if it does not exist.
- Pass the PAG and the proxy to the file system to establish the owner (since the Proxy was also passed to the file system in its creation, any user with a valid proxy can add its PAG).
- Run the job having as working directory the mounting point of the file system (run as the unprivileged user) .
- Ask the file system to remove the PAG from its memory.

Unmounting and destroying the file system

This operation is performed by the same Unix account that requested the creation of the virtual file system and is triggered by the batch system at the end of the job's execution.

PERFORMANCE

Since the file system depends on FUSE, the performance of ScratchFS is mostly determined by FUSE's performance. However, some strategies have been implemented to improve the overall performance of ScratchFS, as follows:

Input/Output Cache

The most important operations for us were *read()* and *write()*. Improving the reading was not difficult since FUSE provides an internal cache that allows large reads for kernel versions before 2.6 (and Linux controls it for versions after the 2.6). The cache mechanism for the write operation uses a linear buffer to collect all the write requests and writes all the buffered data to the disk when the buffer reaches a given size, this gives the idea of a big block size for data writing (if the operation is not sequential the cache is flushed).

Security control

Since verifying grid certificates (grid proxies, in our case) for every single file system operation is expensive, we opted to do it only once, when the virtual file system is created. At this time, a specific thread is created for checking that the grid proxy certificate is valid. When the grid credential expires, all the processes associated to it will lose the ability to access the file system. This mechanism allows us to check the proxy only once per job (at creation time), which is a considerable improvement.

When the access to the file system is based on Unix accounts, a similar mechanism is used, but it only compares the Unix user account requesting access to the file system against the list of authorized users established at the creation of the file system.

Benchmarks

It was desirable to know how ScratchFS behaves compared to the underlying file system, for this we measured the input/output rates in both systems but in addition we compare them to a FUSE throughput file system, to see if the impact in performance is due to the new additions or is inherited from FUSE.

Bonnie++ Bonnie++ [10] has a long tradition as an input/output benchmark so we decided it to use it as the principal indicator of our file system performance. The figure 2 is the main image, since it gives us the writing, rewriting and reading speed of normal operations. This figure shows how small is the difference between ScratchFS and the underlying file system (Ext3), but surprisingly, also tell us that ScratchFS is faster than the throughput file system *Fusexmp*, this is due to our cache strategies for input and output. Note that in the case of the reads we enabled the *large reads* option in ScratchFS but not in Fusexmp.

The interesting point is that the two more important operations for our case (handling relatively big files) are the ones with the best behavior. In the case of a writing per block, the speed is 91.4% compared to a normal Ext3 writing and in the case of per block reading, the comparison gives us 75.7%. Such numbers are quite reassuring for a virtual file system and we think that based on that, the price to pay is not really high compared to the obtained functionality.

When analyzing the CPU overhead we found that using ScratchFS consumed less processing power than a normal call to the system. This was a bit strange, specially because we *know* that the file system does additional things as well as calling the underlying operations, but the answer is quite simple, since bonnie++ only measures the CPU used by its processes, it does not include the overhead of our extra layer (which is precisely what we wanted to know), for this reason bonnie++ is not accurate when measuring the CPU used by the virtual layer.

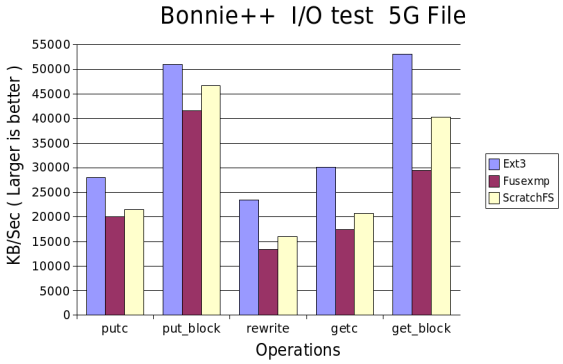

Figure 2: Bonnie++ benchmark

Kernel Compilation Time Although bonnie++ is more suitable to test our file system (we expect CPU intensive applications that will transfer large quantities of data), the comparison would not be complete without a more *real world* type example. We decided to measure the time needed to compile the Linux kernel, which at the moment has 18434 files and 1120 directories.

We used the *time* command to measure the time needed by the compilation and we will only talk about the *real* time given by the wall clock. This might be slightly misleading since different tasks could had used the CPU in that time but since we *guarantee* that only ScratchFS and the compilation are being executed at the moment, it is an acceptable assumption.

In the figure 3, we can see the time taken by the compilation using the three different file systems. The overhead is due to the security check while processing each one of the files and directories, it is also caused by the small size of the files in the kernel tree (97.34% of the files are less than 64K), which makes the cache efforts almost useless. Even in such scenario, we can see that the general lost of performance is around 33%.

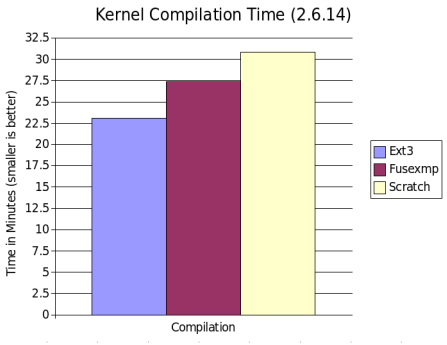

Figure 3: Kernel Compilation

CONCLUSIONS

We have developed a virtual file system that works as a sandbox for input/output operations. Although many sandboxing utilities have been proposed, we think our project presents an interesting functionality that bases the security on grid identities without the need of a one-to-one mapping to local identifiers. Although this is only one of the characteristics, it also provides ideas for a native Grid File System that could implement the access control using only grid identities. Such implementation in kernel side is not trivial and more research has to be done to see its viability.

Even without GSI capabilities, a sandbox for grid jobs is still useful. Seen as a complement to the batch scheduler, it can control the used space and the access in a more detailed way. It can also report information on the real utilization of the scratch space and it establishes a generic implementation that can be easily modified to include additional operations (for instance, an auditing system).

REFERENCES

[1] I. Foster, C. Kesselman, eds., *The Grid 2: Blueprint for a New Computing Infrastructure* Morgan Kaufmann, San Francisco, CA, November, 1999.

[2] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, *A secure environment for untrusted helper applications (confining the wily hacker).* Proceedings of the Sixth USENIX Security Symposium, July 1996.

[3] D. Peterson, M. Bishop and R. Pandey *A Flexible Containment Mechanism for Executing Untrusted Code* Proceedings of the 11th USENIX Security Symposium, August 2002.

[4] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, O. Lodygensky. *Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid* FGCS Future Generation Computer Science, 2004.

[5] D. Wagner, *Janus: an Approach for Confinement of Untrusted Applications* EECS Department, University of California, Berkeley, 1999.

[6] M. Callman and P. Machek, *Subterfugue : a framework for observing and playing with the reality of software* http://subterfugue.org, 2004.

[7] K. Keahey, K. Doering and I. Foster, *From Sandbox to Playground: Dynamic Virtual Environments in the Grid* Proceedings of 5th International Workshop in Grid Computing (Grid 2004), Pittsburgh, PA, November 2004.

[8] http://fuse.sourceforge.net/

[9] http://www.globus.org/security/

[10] http://www.coker.com.au/bonnie++/

[11] http://www.gridsite.org/slashgrid/