Launching large computing applications on a disk-less cluster

Rainer Schwemmer1, Juan Manuel Caicedo Carvajal2 and Niko Neufeld3
1 2 3 European Organization for Nuclear Research (CERN), CH-1211 Geneva 23 Switzerland
E-mail: rainer.schwemmer@cern.ch

Abstract. The LHCb Event Filter Farm system is based on a cluster of the order of 1,500 disk-less Linux nodes. Each node runs one instance of the filtering application per core. The amount of cores in our current production environment is 8 per machine for the old cluster and 12 per machine on extension of the cluster.

Each instance has to load about 1,000 shared libraries, weighting 200 MB from several directory locations from a central repository. The repository is currently hosted on a SAN and exported via NFS. The libraries are all available in the local file system cache on every node. Loading a library still causes a huge number of requests to the server though, because the loader will try to probe every available path. Measurements show there are between 100,000-200,000 calls per application instance start up. Multiplied by the numbers of cores in the farm, this translates into a veritable DDoS attack on the servers, which lasts several minutes. Since the application is being restarted frequently, a better solution had to be found.

Rolling out the software to the nodes is out of the question, because they have no disks and the software in it’s entirety is too large to put into a ram disk. To solve this problem we developed a FUSE based file systems which acts as a permanent, controllable cache that keeps the essential files that are necessary in stock.

1. LHCb High Level Trigger farm architecture

1.1. Hardware and Storage back-end

The LHCb experiment uses a cluster of disk-less servers to filter the events received from the detector that are interesting for physics analysis. This cluster currently consist of about 1,500 Computing nodes with about 13,500 CPU cores and is called the High Level Trigger (HLT) farm [1]. While most of the machines in the cluster do have hard drives, it was decided not to rely on them, because it would be too much maintenance overhead to cope with failure rates of such a large number of drives[2]. As a result all operating system files and also the main application are loaded onto the machines via the network.

The central file server back-end of LHCb consists of a fiber channel SAN attached to 4 main servers. These servers are configured as a High Availability cluster, running in Active-Active mode to share the total load of the entire Online cluster and to minimize down time due to failures. The servers run a set of cluster file systems on top of the SAN. These file systems are used for the DAQ, as well as for shared user and group directories and also for the central software repository. This repository hosts several versions of the HLT application and the necessary calibration and conditions data. The entire repository is about 1 TB in size.
The data itself is striped over two RAID6 sets of ten 7k RPM Disks. To improve metadata performance and safety, the FS metadata is stored on a separate RAID set of 10 Solid State Drives (SSDs), without Write Behind caching. The SSDs can handle in the order of 10,000 IOs per second. No apparent performance degradation could be measured after switching off the caches in the SAN controllers. From these servers the file systems are then exported via NFS and Samba to all the other machines inside the Online network.

1.2. HLT application design
Unfortunately the Application does not yet take advantage of application level parallelism and is essentially single threaded. It was also found, that hyper threading technology can improve the amount of events that can be processed in parallel per farm node by a substantial amount. As a result about 20,000 instances of the application are running during physics data taking on the cluster.

Modular design was one of the key factors during the design phase of the application. The Gaudi [3] framework gives the programmers of the HLT program the ability to produce independent modules of code. This code typically contains event reconstruction or filtering algorithms, which are tied together to decision trees or lines via a configuration description provided to Gaudi. The resulting code base is a collection of many shared object files, scattered over many directories that contain the different sub-trigger projects. During start up, the configuration description tells Gaudi which particular libraries to load. As a consequence, it is not known at the time of software deployment, which of the many files are necessary and which not.

The advantages of this architecture are immediately clear. To change the trigger layout it is only necessary to change the configuration description and restart it. Unfortunately this design has severe drawbacks from a low level, operating system point of view. There are presently more than 200 shared object and more than 100 python modules scattered over about 70 directories. Every module has to be searched by the operating system within all the directories until found. This means that there are on average 35 failed searches per file.

2. RFS as a solution for reducing costly NFS accesses
While it would be possible, in theory, to deploy the software to a RAM disk on the compute nodes themselves, the unpredictable behavior of different trigger configurations and the sheer size of the software repository prevents this. Consequently the software has to be loaded via NFS. Every load request causes NFS to walk the path of the requested module sub-directory by sub-directory, checking if the directory and in the end the file exists. For example, the request for /a/b/foo.so produces the nfs calls:

- stat /a
- stat /a/b
- stat /a/b/foo.so
- open /a/b/foo.so

Since all of the directories within the list of available search paths do exist, it will always be the last entry which is failing. In total this leads to about 40,000 NFS requests per application instance per launch and most of them will fail due to the above mechanism.

2.1. Previous improvement attempts
Several improvements were made to remedy the situation as seen in figure 1 and described in [4]. A first approach was to re-order the list of search paths to have those with higher hit frequency further up in the list than those with less. The improvement was about a factor of...
25% in the total number of NFS calls. This works only up to a certain point though, because some paths are meant to override others in case of more recent developments. This implies a certain order for some paths. Another 25% could be gained by moving the python modules from plain files inside directories into one archive per module. This reduces the number of individual files that have to be searched and opened. In the end the scalability issue still remains and each of the above improvements was made obsolete every time the farm was increased in size.

![Figure 1. File system access system calls for different setups](image)

2.2. RFS

To improve the situation permanently, a proxy like file system was devised, using FUSE [5] as a framework. FUSE allows the creation of arbitrary file systems by providing a set of hooks into the linux Virtual File System (VFS) layer and mapping them into user space. The user writes a simple application which implements callbacks for typical file system operations like stat, open, close, etc. Whenever an operation on the FUSE file system is performed, the calls are then forwarded to the user level program, and the results are presented to the VFS layer. This allows for example to directly map Simple Network Management Protocol (SNMP) agents to be mapped directly as files into the Linux file tree or more sophisticated file systems like NTFS under Linux.

The idea of the RFS file system is to present an image of a different, underlying, file system and listen to the operations performed by user programs. All requests and their answers are memorized, and if an identical request arrives, it is answered directly by RFS instead of asking the mirrored FS. This includes especially requests, that would fail, because a file does not exist in the mirrored FS. Additionally, if a file is opened for reading, RFS makes a copy from the original source and stores it on local storage. In our case this is a RAM disk based on linux-tmpfs which is currently configured with a size of 2 GB. Subsequent reads from that file will then be redirected to the RAM disk instead of the underlying FS. The advantage of running programs like this is, that the code pages of the program are already in memory via the RAM disk and are not duplicated when the program is running. I.e. there is no memory wasted by caching the program executables this way.

Figure 2 shows the internal working of RFS. It keeps a single hash map of paths vs. file stat structures that are saved whenever a request is made. The hash table approach makes the look-up of paths very fast. Whenever a directory is read, a kind of look ahead is performed by scanning the contents of that directory. This will allow to give a quick answer to any call for a file that is not there immediately instead of having to go to the underlying file system one more time.

Of course this whole scheme is not without drawbacks. It only works well if the underlying file systems never changes, or if changes are made to paths, which have not been requested
before. For this case it is possible to use the unlink system call, typically associated with the rm command, to remove the information about a particular file or directory from the file system and trigger a refresh of the cache the next time this node of the FS tree is requested.

It also does not solve the problem of the initial population of the cache, when the FS is mounted freshly. Since there are many instances of the HLT application running on the same node, only the first instance has to populate the cache, while any subsequent instance can already profit from it.

It would probably be possible to achieve a similar behavior tuning the NFS cache of the compute nodes. The lifetimes of the cache that are necessary here though are in the order of hours or days, and there is not really as much control possible from the user side about what gets cached and what not. It is also not easily possible to trigger a refresh of the cache on will, which makes our solution superior to playing around with these tuning parameters. Especially since it is possible to manipulate the cache without root level privileges.

In order to quantitatively determine our progress in start up speed, we have also developed tools to measure precisely how long it takes for a certain configuration to start up [4]. In the frame of a summer student project, a program was written, that directly taps into the LHCb Experiment Control System (ECS) [6]. The program automatically allocates different numbers of farm nodes, performs a complete start up and measures the time until all HLT instances are ready. When a certain number of node configurations has been scanned, a plot similar to fig. 3 is produced, showing total time until all instances are ready vs. the number of running instances. This allows to conveniently monitor the progress we make in start up times, not only for RFS, but also for other improvements made directly to the code base of the program.

3. Results
In 3 the difference between the configuration times before and after RFS was deployed is shown before the last stage of the farm upgrade. Up to approximately 4000 instances, there is not much of a gain visible. From 4000 onwards, the NFS servers are starting to become overloaded and the configuration time for NFS explodes up to almost an hour. This is mostly due to more critical failures and timeouts that have to be fixed by manual intervention. Still this time has to be taken into consideration. If this operation had to be performed by the shift crew without the experts around, the wall clock time for the start up would be even worse.

One can also see, that the scaling is still not perfect for RFS. This is mostly due to accesses of the HLT application to a common database which is not kept in the software repository yet.

For the final upgrade of the farm, additional improvements where made to the code base of the application itself. The startup now causes child processes to be spawned off of one fully
configured application instance. This brings down the number of processes that have to be configured and the configuration time to the number of farm machines, which is in the order of 1500.

4. Conclusion
We have developed a proxy like, FUSE based file system to improve the start-up time of the LHCb HLT application. While this was the original intention of the project, it is not limited to LHCb. It can be used as a general tool to roll out software to a farm cluster without much overhead in packaging and management on the fly, by just starting the program once. The FS caches all common file system calls, especially failed ones, and answers instead of the underlying file system.

At the same time it can copy opened files onto local storage to open them directly from there instead of having to go to the common repository server. In combination with a RAM disk this can cut down load times for huge applications dramatically. In our case we gained a factor of about 10, from almost 1 hour to 5 minutes without having to change the application itself. At the same time this does not cause much memory overhead, because the files that are copied to the RAM disk would have been loaded into memory anyway for program execution.

Future improvements have to be made to improve the initial population of the cache. It is foreseen to implement a mechanism that allows to populate the cache from an input file that contains the necessary list of files. This list has to be created once and loaded into RFS every time the program configuration changes.

References
[1] M. Fran et al., “The LHCb high level trigger infrastructure,” J. Phys.: Conf. Ser. 119 022023
[2] B. Schroeder, G. Gibson, “Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?,” in Proc. of 5th USENIX Conference on File and Storage Technologies, 2007, pp 1–16.
[3] G. Barrand et al., “GAUDI: The software architecture and framework for building LHCb data processing applications,” in Proc. of CHEP 2000, 2000, pp. 45–55.
[4] J. Caicedo, N. Neufeld, R. Schwemmer “Optimising the HLT Farm at the LHCb Experiment,” in Proc. of 17th IEEE NPSS Real Time Conference 2010, 2010.
[5] File system in user space. [Online]. Available: http://fuse.sourceforge.net/
[6] F. Alessio et-al., “The LHCb Run Control,” J. Phys.: Conf. Ser., vol. 219, p. 022009, 2010.