FERMILAB-CONF-12-289-CD

Scalability and Performance Improvements in the Fermilab Mass Storage System

Matt Crawford, Catalin Dumitrescu, Dmitry Litvintsev, Alexander Moibenko, Gene Oleynik

Fermi National Accelerator Laboratory, P.O. Box 500, Batavia, IL 60510-5011, USA

E-mail: {crawdad,catalind,litvinse,moibenko,oleynik}@fnal.gov

Abstract. By 2009 the Fermilab Mass Storage System had encountered two major challenges: the required amount of data stored and accessed in both tiers of the system (dCache and Enstore) had significantly increased and the number of clients accessing Mass Storage System had increased from tens to hundreds of nodes and from hundreds to thousands of parallel requests. To address these challenges Enstore and the SRM part of dCache were modified to scale for performance, access rates, and capacity. This work increased the amount of simultaneously processed requests in a single Enstore Library instance from about 1000 to 30000. The rates of incoming requests to Enstore increased from tens to hundreds per second. Fermilab is invested in LTO4 tape technology and we have investigated both LTO5 and Oracle T10000C to cope with the increasing needs in capacity. We have decided to adopt T10000C, mainly due to its large capacity, which allows us to scale up the existing robotic storage space by a factor 6. This paper describes the modifications and investigations that allowed us to meet these scalability and performance challenges and provided some perspectives of Fermilab Mass Storage System.

1. Introduction

The Fermilab data storage system is the primary storage system for all experiments at the lab and for affiliations. It consists of two major components, dCache, the disk component, and Enstore, the tape component. The storage system manages tens of Petabytes of data and provides data for hundreds of user applications running on hundreds of client nodes. As the capacity of stored data and the number of clients grow the scalability of these systems needed improvement in performance and capacity. This paper describes how different aspects of scalability and performance were addressed and implemented. The work described is the culmination of several years of work.

2. Enstore Performance

All enstore servers were implemented as single threaded Python programs for simplicity, and use connectionless UDP for communications to provide backpressure under heavy loads. This worked quite well for Run II Tevatron data storage and access, although we encountered intermittent problems during high request rates. When US-CMS T1 stress testing began in 2009, bigger performance issues were uncovered. The main issue was that the servers could not cope with the increased rates of incoming requests from hundreds of clients running on hundreds of network nodes. As a result network UDP buffers were becoming overrun, and as the packet drop rates

Operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy.

increased, response times became unacceptable. Implementation of UDP receivers in separate threads provided some improvement, but was not a complete approach.

The version of Python that Enstore was initially programmed in did not have a threading model that was capable of taking advantage of multicore hardware. Threading basically took place in a single process on a single core. However, starting with Python 2.6, standard Unix Inter Process Communications was supported, enabling us to efficiently utilize multiple cores.

The dispatcher of client requests (Library Manager) was modified from single threaded server serving all requests via single port as shown in figure 1 to multi-threaded, multi-processed server serving request of different types via 3 dedicated ports and corresponding processes as shown in figure 2.

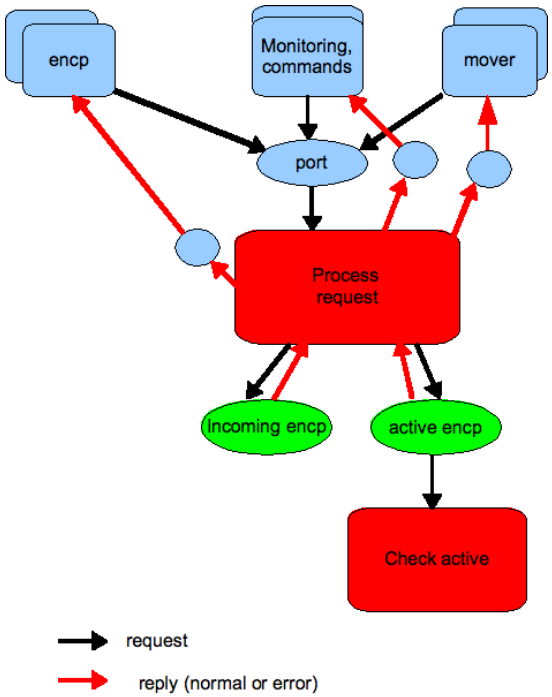

Figure 1. Original Library Manager Implementation.

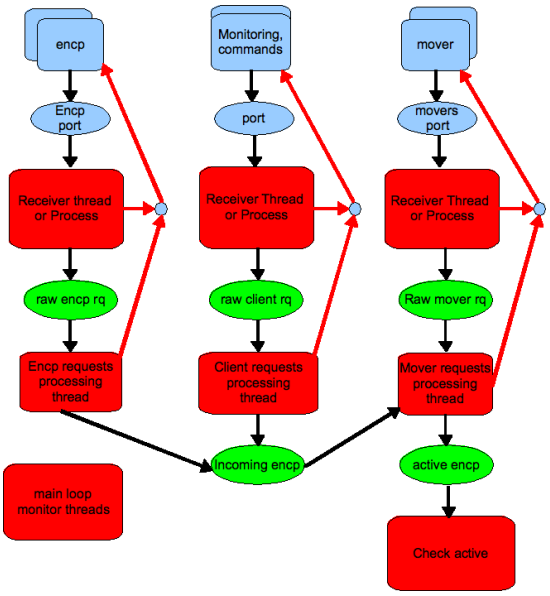

Figure 2. New Multi-threaded Library Manager Implementation

Other servers did not need such drastic modification. They were modified to launch a separate thread for processing some time consuming requests.

Another very important modification was a change in the algorithm for the selection of a request from the queue of pending requests. This algorithm makes sure that no more than a given number of requests from the same client node is active. Without this algorithm the network transfer may become oversubscribed. This algorithm was becoming ineffective with the growth of pending request queue, Improvements were made to more efficiently process lists of nodes.

The priority algorithm was also modified. The initial design of Enstore allowed users to specify the the delta time and delta priority for individual requests. This required the Library Manager to loop over every queued request, calculate its new priority, and then resort the request queue in order to select the highest priority request from the queue. Processing time was $T * N$, where T - time for processing a single request, N - number of requests in the queue for a given group. It was determined that experiments really didnt need this flexibility, so this feature was dropped and the request selection greatly simplified. The new algorithm changes priority of all

pending requests for the same group with the same delta time and delta priority, significantly reducing the processing time for servicing a request.

As a result of all these changes we are able to keep up with high rates of incoming client request as evidenced by figure 3, which shows the request packet drop rate for the original Python threads and the process based threads.

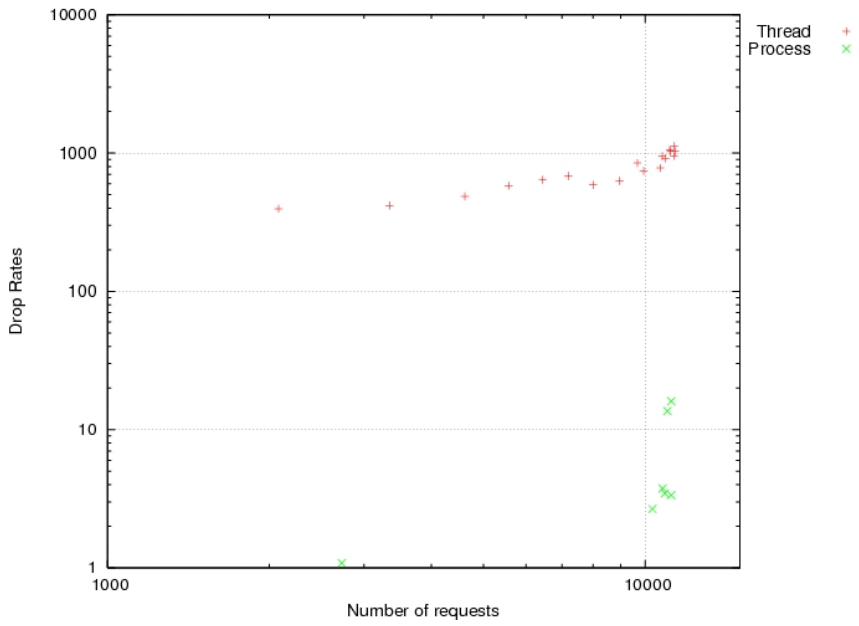

Figure 3. Request Packet Drop Rates

Before this changes enstore was barely able to keep processing 2000 requests in the queue and tens of client requests. After modification it keeps and effectively process 30000 and more requests in Library Dispatcher queue and hundreds of client requests.

3. Storage Resource Manager

The Storage Resource Manager (SRM) [1] is a web service protocol to manage data stored in heterogeneous underlying storage systems on the grid. SRM was selected by the Worldwide LHC Computing Grid (WLCG) as a management protocol to data storage systems that are components of the WLCG Data Grid. It has been agreed that data storage systems accepted by WLCG must implement SRM interface. dCache [2] is one of the storage systems chosen by the majority of LHC Tier 1 centers. dCache SRM interface has been implemented by Fermilab and was funded by DOE and US CMS.

Since dCache SRM is a single point of remote entry into the system, its performance and reliability are critical to successful operation of dCache based LHC Data Storage Systems. Initially SRM was envisioned to be used for WAN only transfers with relatively low rates. The CMS experiment, for example, had set a requirement of 2 Hz for SRM transfers to and from Tier 1 Storage Element [4]. The actual usage exceeded that, but reached a ceiling of about 5 Hz. At one time CMS Tier 1 considered using SRM for LAN transfers for uniformity raising the requirement of transfer rates to 100 Hz or higher. Apparently SRM dCache could not satisfy this requirement.

A review of SRM dCache [3] has identified the following performance issues:

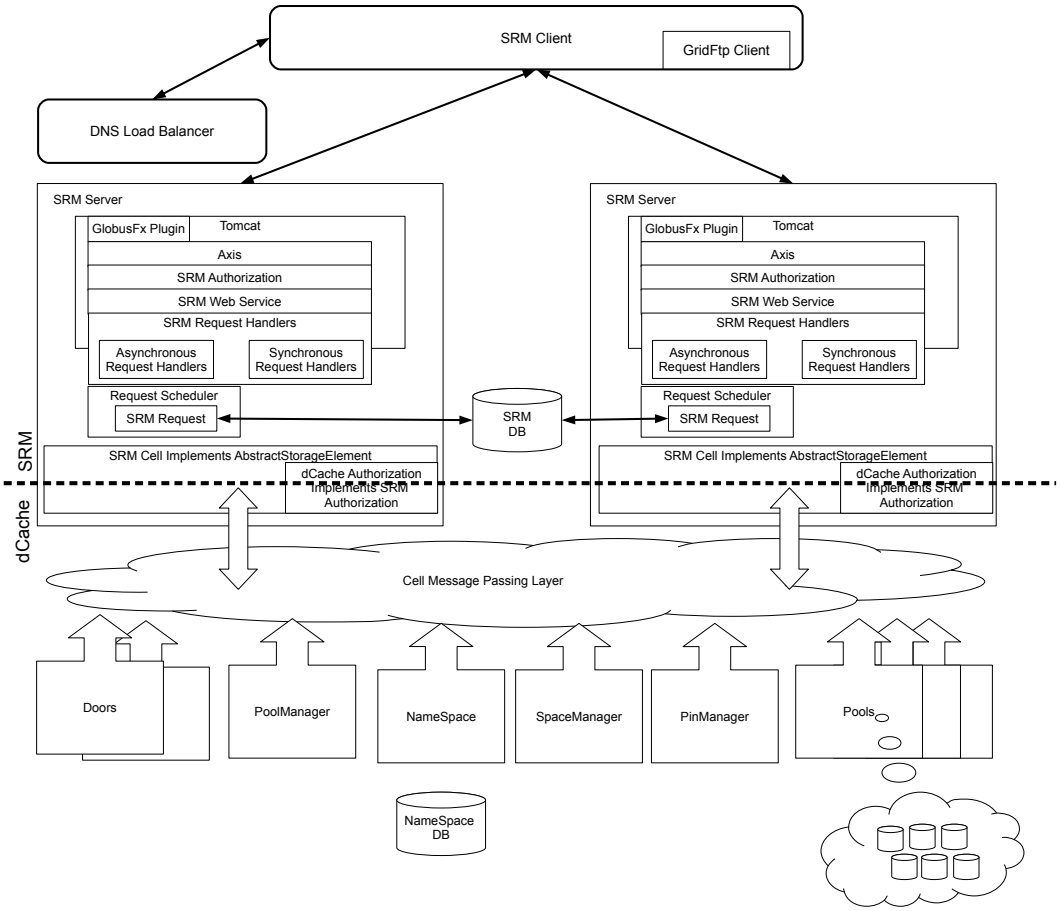

Figure 4. Scalable SRM architecture.

- high CPU load due to GSI authentication and credential delegation. This has been mitigated by introducing caching of public/private key pairs used in GSI authentication in jglobus code (work performed by NDGF). This change reduced CPU consumption by GSI handshakes from 97% to about 40%.
- unscheduled blocking functions, mainly `srmls` and `srmrm`, created high load on SRM server and namespace provider causing starvation of network connection slots due to high volume and long duration of these operations. This was solved by introducing scheduled `srml` requests as SRM protocol allows asynchronous processing of `srmls` requests. Unlike `srmls` the `srmrm` function is blocking. Bulk `srmrm` removals, while blocking, were throttled by internal scheduling of removal requests to namespace provider in smaller, configurable batches.
- SRM dCache is a single point of failure. A horizontally scalable SRM solution was pursued using Terracotta to manage distributed object store to share SRM request state across multiple SRM servers. The results were presented at CHEP 2010 [4] and were found to be discouraging as the gains from the multiple parallel SRM server nodes were almost entirely offset by synchronization overheads introduced by global locks [4].

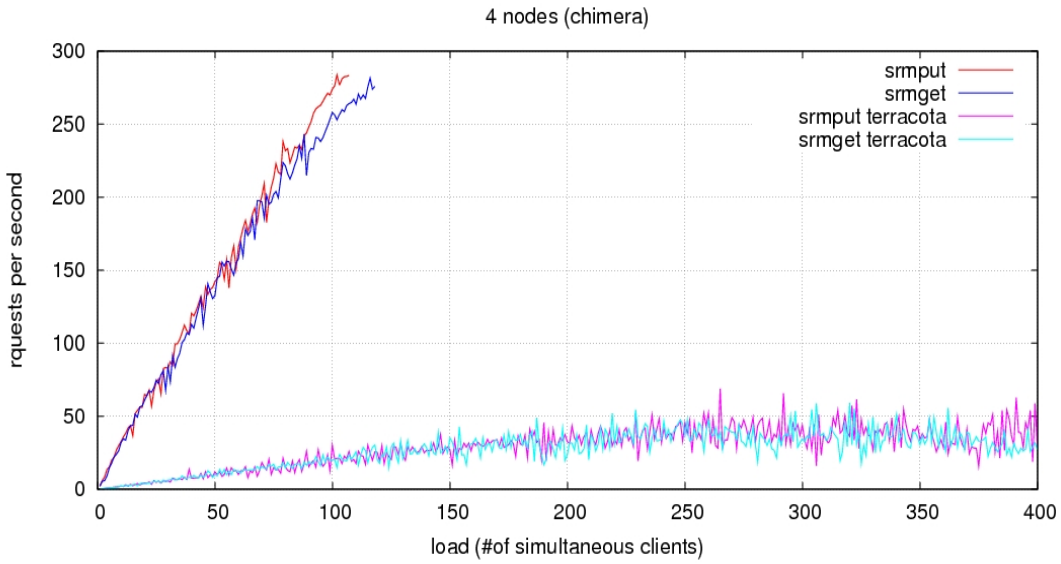

Figure 5. Scalable SRM performance

This paper describes continuing efforts to distribute SRM servers that relies on custom solution that utilizes already existing SRM database that stores SRM requests to keep requests' states and share them between the servers.

The basic principle used to make SRM scalable was to allow multiple instances of SRM servers to run in parallel on different hardware. A DNS load balancer distributes the SRM requests in a round-robin order to the set of SRM servers.

Each SRM instance shares a common database backend for request state persistency. The instance of SRM that received the request serves it. Other SRM instances provide the status of this request by restoring it from the database without caching. Any action for a request starts with obtaining the request status from the database. Each SRM instance periodically polls in-memory requests to re-check their status in the database to detect request status changes that might have been processed by the SRM instance that actually did not schedule the request. The architecture of scalable SRM cluster is depicted in figure 4.

The plot in figure 5 shows request processing frequency vs. number of simultaneous clients for srmput and srmget operations when running 4 SRM instances. The result is compared to terracotta based scalability solution, which was contemplated in 2010 based on SRM code and performance review [3]. Terracotta behaved poorly because of latencies introduced by global memory locks. This is not an ultimate judgment of Terracotta distributed memory management, but rather a testament to the fact that SRM code as designed is not easy to adapt use Terracotta efficiently.

CMS Tier 1 currently opeartes two SRM servers in production. The load is spread evently across two nodes. Since SRM is used exclusively for WAN transfers the rate never approaches 100 Hz. The service availability has improved as become possible to conduct rolling maintenance without service interruption.

4. Capacity Planning

We have LTO4 tape drives in robotic storage. Adding new types of media, tape drives and software support for them provides additional capacity. We evaluated and certified LTO5 technology, whose capacity of 1.5 TB would have given us almost a factor of two gain in capacity over LTO4 [5]. At about this time, Oracle came out with their new T10000C series enterprise class drives, which can provide up to 5.4 TB in capacity. We also certified Oracle T10000C tape drives and corresponding media. We have decided to adopt T10000C, mainly due to its large capacity, which allows us to scale up the existing robotic storage space by a factor of 6.75 over LTO4.

Replacing existing technologies with new increases demand for reliable and scalable solutions for migration of data from old media to new. We are quite successfully coping with this problem currently using dedicated migration nodes. This is an on-going process that has gone from 9940A to 9940B to LTO2&3 to LTO4 and now to T10000C.

Another very important issue is ineffective treatment of small files. We have addressed this issue with the Enstore File Aggregation feature which introduces a temporary storage of small files in dedicated internal cache. These files then get aggregated in a bigger container based on specified policy and migrates this container to tape. We also take advantage of the File Sync Accelerator Feature that the T10000C drives offer for making the writing of small files more efficient, and we have to incorporate buffered tape marks in our migration process in the future. Our overall approach to small files will be reported on in the next CHEP.

5. Conclusion

We have substantially improved the scalability and performance of major software components of the Fermilab data storage system as well as capacity of the Tape Libraries, which allowed us to meet increasing demand in capacity of stored data and servicing user requests. As the next step for performance improvements, we have just put into production the first release of the Enstore file aggregation feature. This feature provides policy based aggregation of small files into larger container files on tape, which allows increased efficiency transferring small files to and from tape.

References

[1] Sim A and Shoshani A 2008 "The Storage resource manager interface specication version 2.2" Tech. Rep. GDF.129 Open Grid Forum URL `http://www.ogf.org/documents/GFD.129.pdf`

[2] dCache URL `http://www.dcache.org`

[3] Oleynik G, Crawford M, Behrmann G, Dumitrescu C, Gysin S, Levshina T, Salgado P and Perelmutov T 2009 "SRM scalability/performance review" CD DocDB 3163 Fermilab URL `http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=3163`

[4] Crawdord M, Moibenko A, Oleynik G, Perelmutov T 2011 *J. Phys: Conf Series* **331**

[5] Oleynik G *et al.* 2011 LTO-5 Certification Test Results CD DocDB CS-doc-4365-v2 Fermilab URL `https://cd-docdb.fnal.gov:440/cgi-bin/ShowDocument?docid=4365`