Access Coordination Of Tertiary Storage For High Energy Physics Applications *

Luis M. Bernardo, Arie Shoshani, Alexander Sim, Henrik Nordberg
Scientific Data Management Research Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{LMBernardo, AShoshani, ASim, HNordberg}@lbl.gov
tel +1-510-486-5171
fax +1-510-486-4004

Abstract

We describe a real implementation of a software component that manages caching of files from a tertiary storage management system to a large disk cache developed for use in the area of High Energy Physics (HEP) analysis. This component, called the Cache Manager, is a part of a Storage Access Coordination System (STACS), and is responsible for the interaction with a mass storage system that manages the robotic tapes (we used HPSS). The Cache Manager performs several functions, including managing the queue of file transfer requests, reordering requests to minimize tape mounts, monitoring the progress of file transfers, handling transient failures of the mass storage system and the network, measuring end-to-end file transfer performance, and providing time estimates for multi-file requests. These functions are described in detail, and illustrated with performance graphs of real-time runs of the system.

1 Introduction

Like so many other scientific disciplines, HEP experiments produce huge amounts of data that, given the usual budget constraints, need to be stored in robotic tape systems. For instance, the STAR experiment at Brookhaven National Laboratory that will start collecting data by mid 2000, will generate 300 TB of data over the course of three years. Storing such amounts of data in disks is certainly unreasonable and also a waste of financial resources since most of the data will not be used often, yet they need to be archived. In practice all the data will be stored in tapes and the amount of available disk space will amount to a few percent of the total space needed to store all the data. Given the fact that retrieval of data from tapes is much slower than from disk, the need for smart cache management

*This work was supported by the Office of Energy Research, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

systems, that coordinate both the retrieval of data from tapes and the use of the restricted disk cache, is real [3, 2, 1]. With this goal in mind we developed STACS (Storage Access Coordination System) [4] to be used by the STAR experiment. STACS was designed to take advantage of the fact that the particle collisions, recorded by the STAR measuring devices, are independent of each other, and therefore the processing of each collision's data can be done in any order. This provides the ability to choose the order of caching of data from tape to disk cache, so as to optimize the use of the cache by multiple users. In addition, since we know ahead of time all the files needed for processing for all the users currently in the system, we can order the scheduling of file transfers to minimize the number of tape mounts.

This paper is organized as follows. In section 2, we start by briefly describing the application domain of High Energy Physics and how the particular needs of that domain influenced the design of STACS. We briefly discuss the architecture of STACS, and describe the process of executing queries. In section 3, we describe in detail the component responsible for interacting with the system that manages the tapes (we used HPSS), called the Cache Manager. In this paper, we emphasize many of its features, including the support of a request queue, the reordering of file transfers to minimize tape mount, and the handling of errors and system failures. We conclude in section 4.

2 The STACS Architecture

We describe in this section the components of STACS, and the reasons for the modular architecture of the system. First, we need to describe briefly the application domain, the kind of queries applied to the system, and what is expected from the application's point of view.

2.1 HEP Application Domain

In the HEP STAR experiment, gold nuclei are collided against each other inside an accelerator and the results of such collisions are recorded by a very complex set of measuring devices. Each collision is called an event and the data associated with each event is in the order of 1-10 MB. It is expected that the experiment will generate 10^8 such events over 3 years. The raw data recorded by the measuring devices are recorded on tapes. They are organized in files, each about 1 GB in size. The data then undergo a "reconstruction" phase where each event is analyzed to determine what particles were produced and to extract summary properties for each event (such as the total energy of the event, momentum, and number of particles of each type). The number of summary elements extracted per event can be quite large (100-200).

The amount of data generated after the reconstruction phase ranges from about 10% of the raw data to about the same size as the raw data, which amounts to about 30 - 300 TBs per year. Most of the time only the reconstructed data is needed for analysis, but the raw data must still be available. It is against the summary data that the physicists run their queries searching for qualifying events that satisfy those queries. All queries are range queries (for example, 5 GeV $<$ energy $<$ 7 GeV, or 10 $<$ number of pions $<$ 20). For each query, STACS has to determine which files contain the reconstructed data (or the raw data if they are requested), and to schedule their caching from tape for processing.

Given the fact that the different events (collisions) are independent of each other, it is irrelevant for the physicists whether they receive the qualifying events in the order they were generated or any other order, as long as they receive all qualifying events. So, what the physicists need is a away to map their queries to the qualifying events stored in the tape system and to efficiently retrieve those events from tape to their local disk so that they can run their analysis programs. STACS was designed with this in mind. It is typical that physicists form collaborations, where 10-100 users study the same region of the data. Therefore, there is good likelihood that queries of different users will overlap in the files that they need. STACS is designed to maximize the use of files once they are cached to disk, by striving to make each file available to all application programs that need it.

2.2 STACS

The STACS architecture consists of four modules that can run in a distributed environment: a Query Estimator (QE) module, a Query Monitor (QM) module, a File Catalog (FC) module and a Cache Manager (CM) module. All the communication between the different modules is handled through CORBA [5]. The architecture of the system is shown in Figure 1. The purpose of this paper is to describe in detail the capabilities provided by the CM. However, to put this in context we describe briefly the function of each module next.

The physicists interact with STACS by issuing a query that is passed to the QE. The QE utilizes a specialized index (called a bit-sliced index) to determine for each query all the events that qualify for the query and also the files where these events reside. This index was described in [4]. The QE can also provide time estimates before executing a query on how long it will take to get all the needed files from the tape system to local disk. The estimate takes into account the files that are currently in the disk cache. If the user finds the time estimate reasonable then a request to execute the query is issued and the relevant information about files and events is passed to the QM. The job of the QM is to handle such requests for file caching for all the users that are using the system concurrently. Since the users don't care about the order they receive the qualifying events the QM is free to schedule the caching of files in the way that it finds most efficient (for instance, by requesting first the files that most users want). The QM uses a fairly sophisticated caching policy module to determine which files should reside in cache at any time. The QM marks each file requested by one or more queries with a dynamic weight proportional to the number of queries that still need that file. The caching policy uses this weight to maximize the usage of the cache by queries. Any files that happen to be in cache and can be used by an application are passed to the application as soon as it is ready to accept the data (i.e. when it is not busy processing the previous data). Files are removed from cache only when space is necessary. The files with the lowest weight are removed first. A more detailed description of the caching policy is also given in [4].

After the QM determines which files to cache, it passes the file requests to the CM one at a time. The CM is the module that interfaces with the mass storage system, which in the case of STAR is HPSS. It is the job of the CM to make sure that the files requested

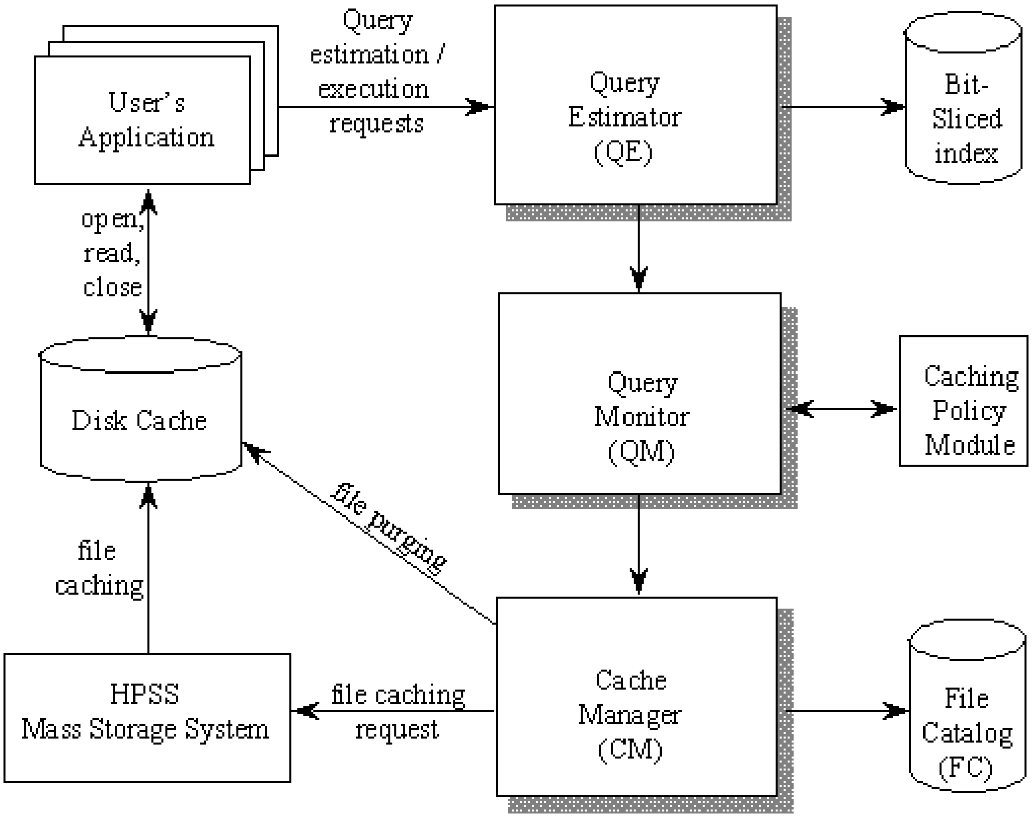

Figure 1: The STACS architecture.

by the QM are properly transferred to local disk. When a request reaches the CM a file is identified by a file id (*fid*), a logical name. To be able to transfer the file from HPSS to local disk the CM needs to convert the file logical name into a real physical name. This mapping can be obtained by consulting the FC, which provides a mapping of an *fid* into both a HPSS file name and a local disk file name (the full path of the file). It also includes information about the file size and the tape id (*tid*) of the tape where the file resides.

To visualize the operation of STACS, we include here a graph of a real run of the system processing multiple files (Figure 2) for a single query. The x-axis represents time. Each jagged vertical line represents the history of a single file. It starts at the bottom at the time it was requested, to the time it was cached to HPSS cache, to the time is was moved to the shared cache, to the time it was passed to the requesting query, and terminates (at the top) after the application finished processing all the events it needs from that file. As can be seen, initially a request for two files was made (one to process, and one to pre-fetch), and only after the first file was processed the application made a request to cache another file.

3 The Cache Manager

The CM performs mainly two functions: it transfers files from the mass storage system (HPSS) to local cache and purges files from local cache. Both actions are initiated by the QM. The transfer of files requires a constant monitoring. The CM performs a variety of

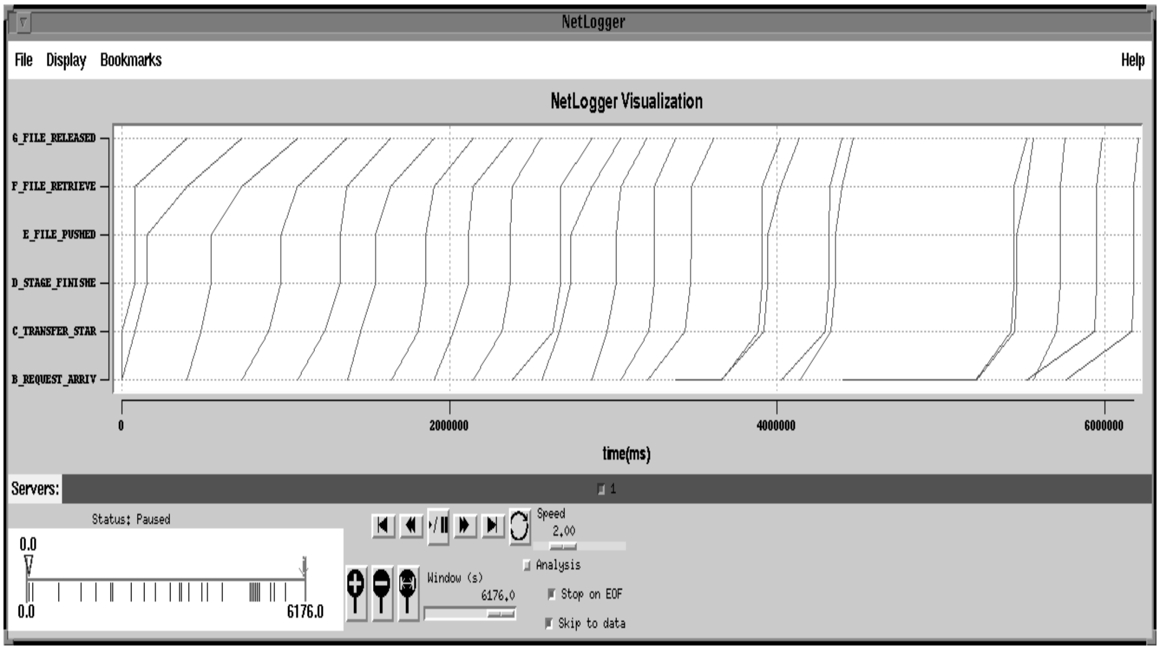

Figure 2: Tracking of files requested by a query.

different actions towards that end. It measures various quantities, such as the transfer rate of each file, it keeps track of the amount of cache in use, and (whenever a transfer fails) it detects the type of failure by parsing the PFTP output looking for errors.

3.1 File Transfers

The CM transfers files from the mass storage system (HPSS) to local cache using the parallel file transfer protocol (PFTP). The CM is multithreaded and can handle many file requests at the same time (in fact, there is a different thread for each PFTP request). Since the number of PFTPs that HPSS can handle concurrently is limited (by the memory available to HPSS) the CM needs to make sure that it doesn't swamp HPSS with too many concurrent PFTPs. This is a required feature because the HPSS system is a resource shared by many users and as such all users have to make sure they don't use more than their share. And even though the HPSS system administrator can block PFTP requests from any user, the system will work better if the users stay within their PFTP quotas. The CM handles this for all its users by queuing the file requests that it receives from the QM and never serving more than the number of PFTPs allocated to it. Thus, STACS and in particular the CM, performs the function of serving its users in a fair fashion, by not allowing any single user to flood the system with too many file caching requests. In STACS the number of allowed PFTPs can be changed dynamically by the system administrator, while the cache manager is running. If this limit is reduced, it simply stops issuing PFTPs until the number of PFTPs in progress reaches the new limit.

3.2 Queue Management

Since the CM builds up a queue of file requests that cannot be served while the number of PFTP requests is at its maximum, opportunities arise for rescheduling the order of requests in the queue so that files from the same tape are asked together one after another. The idea is that the transfer rate from HPSS to local cache will increase if the number of tape mounts is minimized. This is particularly important if the number of tape drives is small and the network bandwidth between HPSS and local cache is large. The goal is to have an aggregated transfer rate as high as possible and that can be achieved by minimizing the "idle" transfer periods during tape mounts. Obviously this gain obtained by rescheduling the queued requests comes at a cost, the cost of bypassing older requests in the queue and instead serving younger requests just because they happen to be from a more "popular" tape. We leave the responsibility of deciding how much rescheduling to do to the STACS administrator and that can be done by dynamically changing a "file clustering parameter" that characterizes the clustering of requested files according to the tape they reside in. Thus, choosing the parameter to be, say, 5 means that if a file from some tape was just transferred to local cache, then on average 4 more files from the same tape will be requested (this only holds true in an infinite queue, but it's a good approximation). Choosing the parameter to be 1 means that no rescheduling will be done and the files in the queue are served in a first come first serve order. Figures 3 and 4 show the order the files were requested versus the tape they reside in for two runs of the same set of queries. The "file clustering parameter"s used were 1 and 10 respectively. The important thing to notice is that in figure 3 there is a constant changing of tapes.

3.3 Query Estimates

One of the most interesting, and also the most difficult to implement, features of the CM is the capability of estimating how long the files needed for a query will take to transfer to local cache. Even though the users get the time estimate through the QE, the real estimates are done by the CM and passed to the QE. The estimates are done by checking which subset of the set of files needed for a query are in cache (call that X), which are already scheduled to be cached, and are in the CM queue (call that Y) and which still have to be requested (call that Z). The CM can use the current transfer rate to estimate how long the files needed will take to transfer. If the current transfer rate happens to be zero, either because no files are being transferred or because the network is temporarily down, then a default transfer rate is used. We describe in Section 3.5 how the actual transfer rates are obtained over time. So far, we used the maximum transfer rates obtained when the system is heavily loaded as the default transfer rate values. In the future, we plan to tune the default transfer rate dynamically, averaging the maximum transfer rates for the last 24 hours (or whatever default period is preferred).

To get a best case estimate, assuming this query gets top priority, we need only to divide the sum of sizes of files not in cache by the transfer rate Tr, i.e. $(s(Y) + s(Z))/Tr$ where $s(Y)$ and $s(Z)$ are the sum of sizes of files in set Y and set Z respectively.

However, we also want to get a realistic estimate. We achieve this as follows. For the X files that are in cache we assume they continue to be available to the application since they

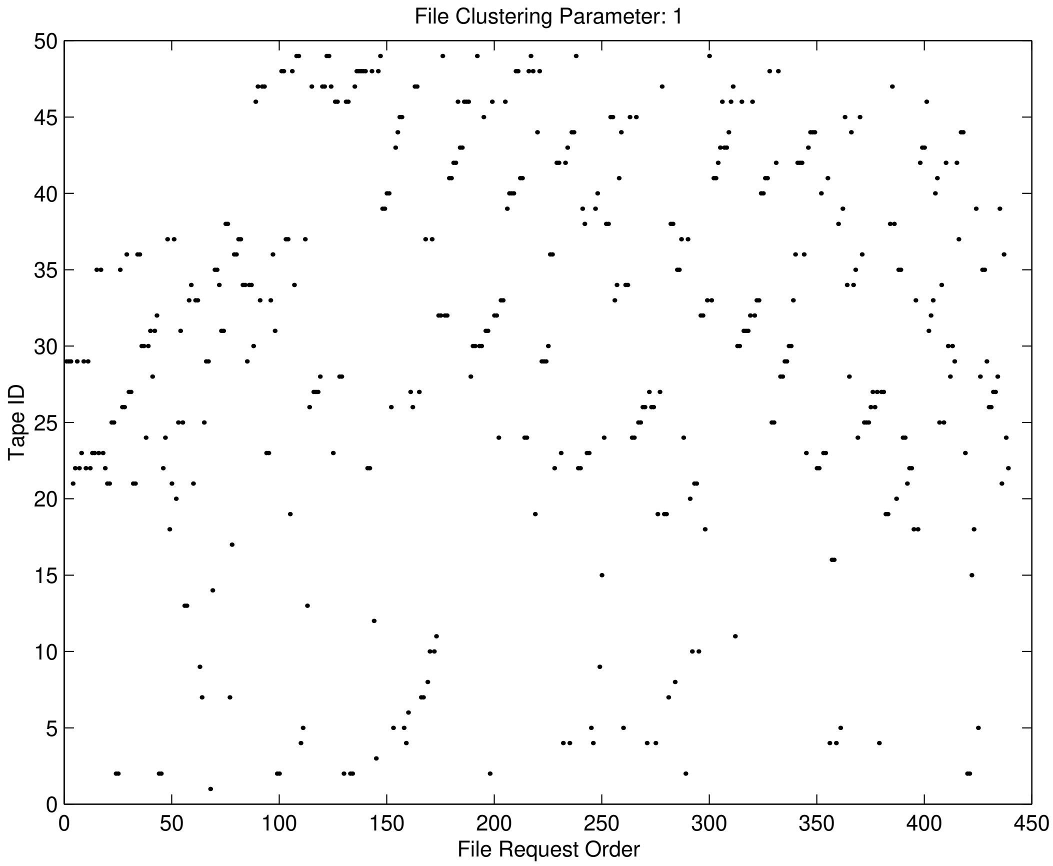

Figure 3: File request order in the absence of file clustering. Files are requested on a first come first serve basis. Each point in the x-axis corresponds to a new file request.

will be marked as needed. For the Y files in the CM queue, we have two cases to consider. If the set Z is empty then we don't need to consider the set of files in the queue that come after all the files in set Y. Call the set of remaining files in the queue Y' (we only need to consider the files in the queue from the first file to the last file in Y). Then the estimate is $s(Y')/Tr$. If on the other hand the set Z is not empty then we need to take into account that all the files in the queue need to be processed before any files in the set Z. We call the set of files in the queue T. Let then the number of queries in the system be q. For our estimate, we assume that each of the queries will be served in a round robin fashion, and that there is no file overlap between the queries. Then for the Z files we need $qs(Z)/Tr$, assuming that all files have similar sizes. So the total time estimate is $(qs(Z) + s(T))/Tr$.

Of course these estimates are only reasonably good if the system doesn't run out of cache space (in which case the file transfers have to stop until some files can be purged) and if the number of queries stays the same during the period that the query in question is being processed. Figures 6 and 5 show a comparison between the estimated time and the real time for the same set of twenty queries run from the same initial state (no files initially cached), with the difference that in one case the queries come 5 minutes apart and in the other case they come 20 minutes apart. In these runs the processing time per event (the time spent processing an event by the application) was chosen very small so that the amount of

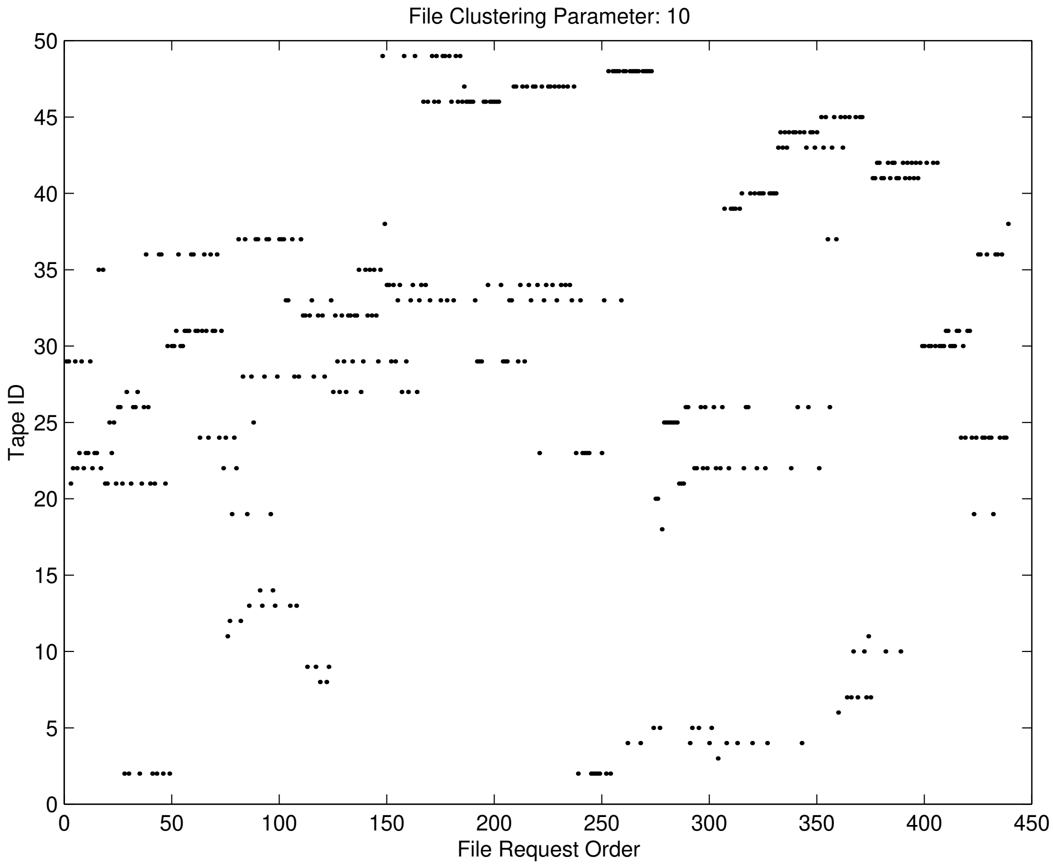

Figure 4: File request order with a file clustering parameter of 10 files per tape. As many as 10 successive requests from the same tape are made if they are found in the queue.

time the QM holds a file in cache is negligible when compared with the transfer time. The queries were designed to complete in about 20 minutes each. Figure 5 shows the estimates when the same set of queries arrive 20 minutes apart. This time is enough to transfer all the files needed by the query before the new query comes in. As a consequence the estimates are very accurate. They are biased towards shorter transfer times because the CM used the default transfer rate to calculate the transfer times, and the default transfer rate was chosen as the maximum transfer rate that the network supports. That default is not sustained for longer periods and hence the shorter time estimates.

On the other hand, in figure 6 the queries arrived 5 minutes apart. In addition, we did not take into account the number of queries that were in the system when a new query started. Since there was not enough time to finish a query before a new query arrives (we chose the queries so that they request approximately the same number of files every time), the requests for files pile up in the CM. This explains why successive time estimates grow larger and larger; the requests for files pile up faster than the CM can serve them. We can also see that the estimates were very poor and fell short of the real transfer times, because the estimate did not account for the number of queries in the system. This gave us the insight to take the number of queries into account, a feature that is now being implemented. We note that even so, the fact remains that when an estimate is done the CM knows nothing about

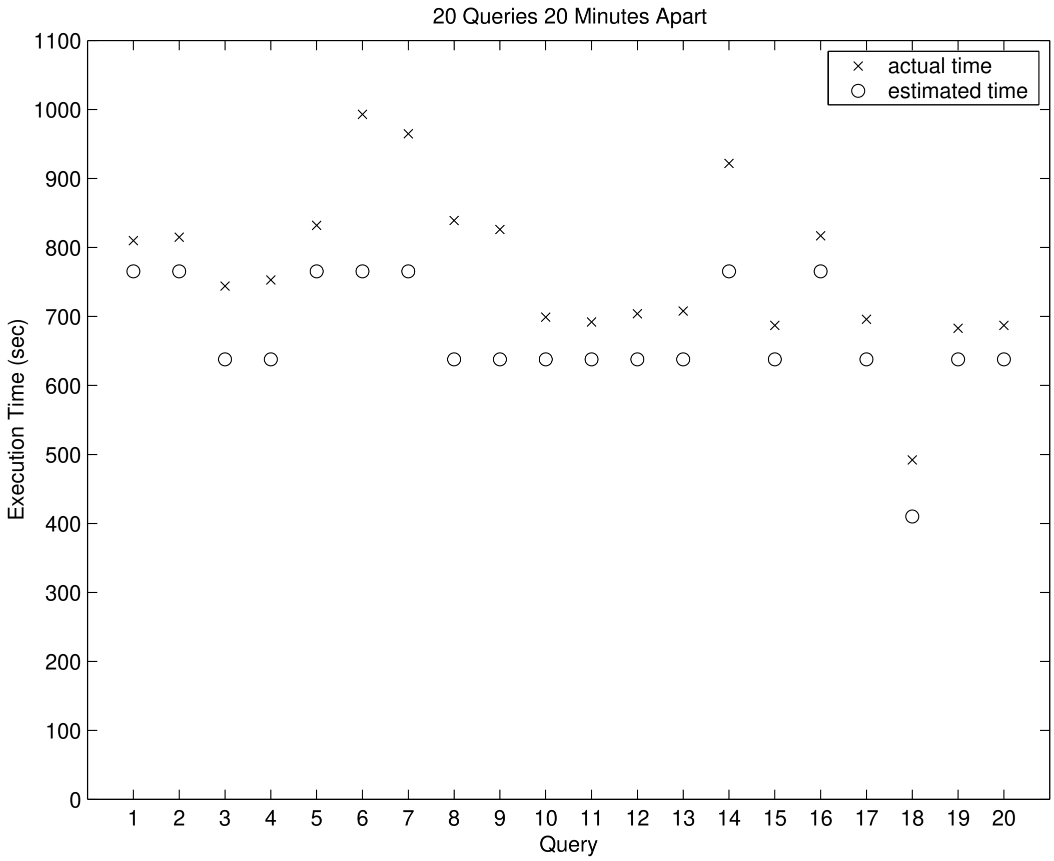

Figure 5: Comparison between estimated time and real transfer time when the queries run alone in the system.

the queries that will come in the future. Because of the round robin policy we currently use, such queries will request some files before all the files for previous queries were requested. Nevertheless, our estimates are pretty accurate since they are based on a measured transfer rate, the files in cache for that query, the number of files in the queue, the actual sizes of files, and the current load on the system, measured as the number of concurrent queries being processed.

3.4 Handling PFTP Errors

The most important functionality of the CM is the handling of transfer errors. Sometimes the PFTP transfer fails, either because HPSS misbehaves or breaks down, or because the network is down or even because the requested file doesn't exist in HPSS. So to make sure that the file was successfully transferred to local disk the CM starts by checking the PFTP output looking for the string "bytes transferred" (this string also appears at the end of a ftp transfer). If that string is not found the CM parses the PFTP output looking for possible error messages, and depending on the result different paths are taken. For instance, if the file doesn't exist on HPSS the CM just reports the fact to the QM. If on the other hand, the transfer error was due to some HPSS error (say, an I/O error) the CM removes the partially transferred file from disk, waits a few minutes, and then tries again to transfer the same file.

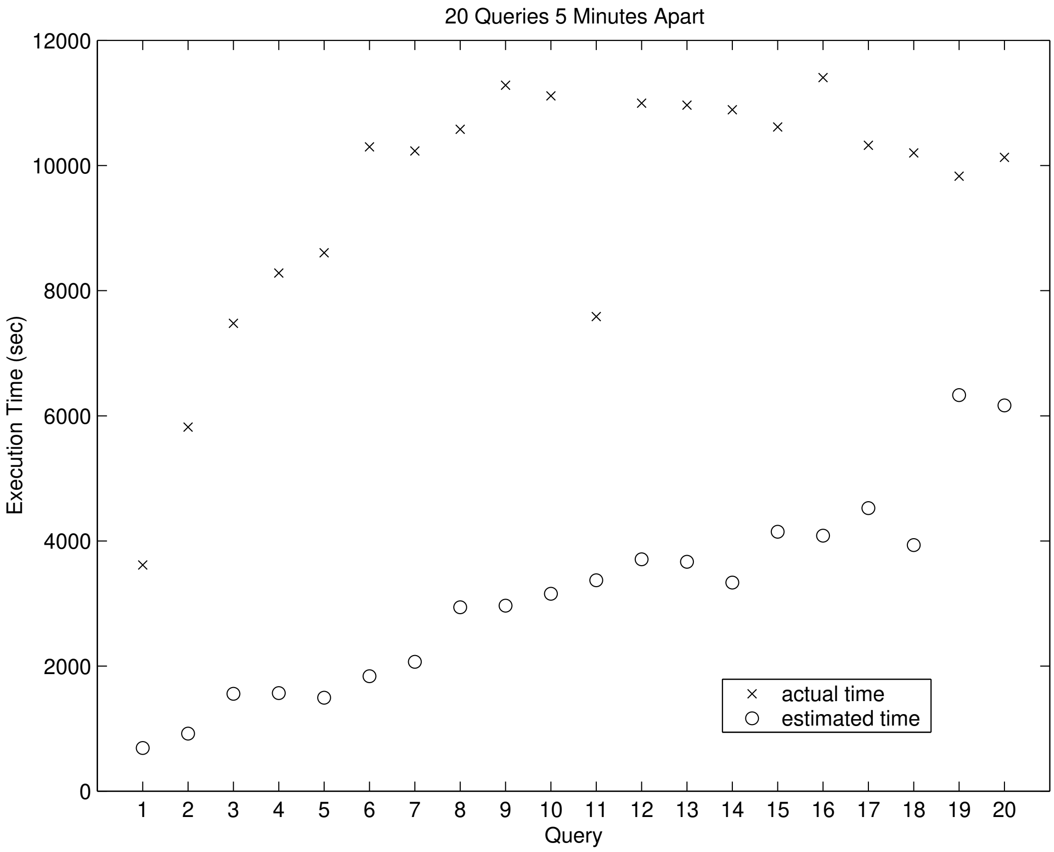

Figure 6: Comparison between estimated time and real transfer time when there is sharing of resources between queries.

This functionality of the CM is very important because it insulates the rest of the system and the user's application from HPSS and network transient failures. All the user perceives is that the file may take longer to cache or that it doesn't exist. This situation is shown in Figure 2. It shows two gaps in the file transfers, one long and one shorter. This was due to an HPSS server failure that was then restored. The CM checked HPSS periodically till it recovered and then proceeded with file transfers.

The possible errors or reasons that cause a PFTP to fail are the following:

- File not found in HPSS. This is an irrecoverable error. The CM gives up and informs the QM.

- Limit PFTPs reached. This happens if other users use more than their share of allocated PFTPs. When this happens it is impossible to login to HPSS. The CM handles this by re-queuing the file request and trying again later.

- HPSS error. Some are recoverable (like an I/O error or a device busy error), others are not (a non existing file, or a wrong read permission). The CM handles the recoverable errors by trying again up to 10 times. This is a default number that can be changed dynamically. The assumption is that if a transfer fails 10 times then something is really wrong with the file. Another approach, which we did not implement, is to

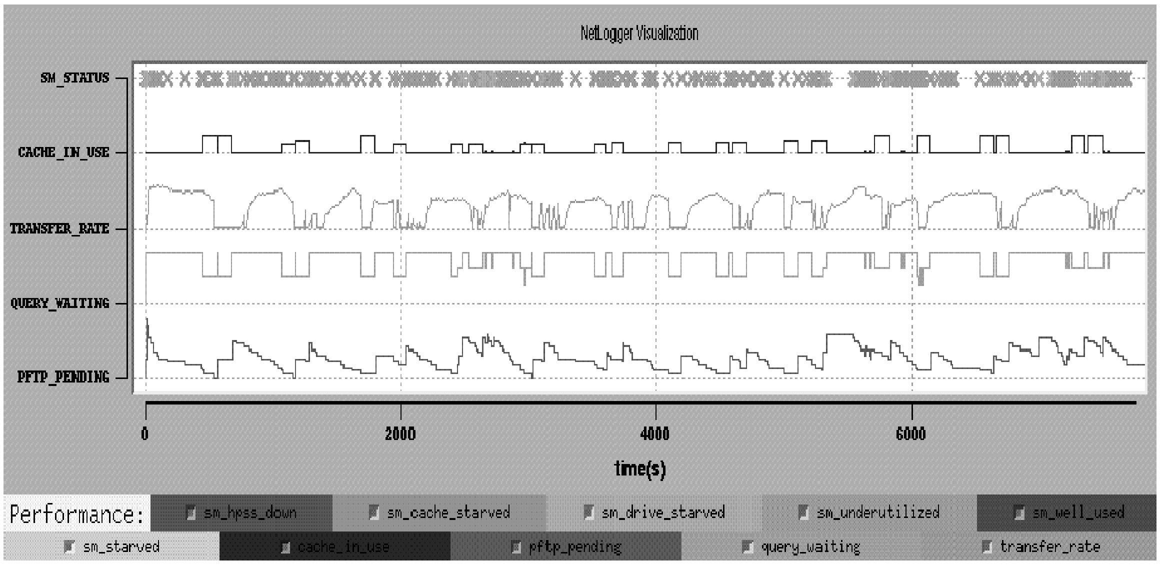

Figure 7: This graph shows several quantities that STACS can display dynamically and that characterize the overall status of the system.

have a timeout mechanism where no more PFTP retrials would be done once the timeout limit was reached.

3.5 Measurements

The CM keeps track of various quantities that characterize its status at any time. One of those, and probably the trickiest one to measure, is the transfer rate between HPSS and local cache. When a PFTP is launched the requested file transfer may not start right away. This is particularly true if the file happens to be on tape instead of being in the HPSS own cache. In that case the tape has to be mounted before the transfer can really start. This fact is not known to the CM. After the transfer occurs the CM can find out how much time was really used in transferring the file and how much time was used in mounting the tape and seeking to the right place on tape, but that information comes too late to be of any use in estimating the instantaneous transfer rate. The CM can give very accurate measurements of the instantaneous transfer rate by following a different approach: it periodically (say, every 15 seconds or whenever a file transfer ends) checks the local size of all the files currently being transferred. By measuring the total number of bytes transferred between now and the previous measurement and the amount of time elapsed, it can give an accurate value for the transfer rate. To smooth out quick fluctuations, it gives a moving average of the transfer rate measured over the last, say, 10 measurements.

Other quantities the CM keeps track of are the number of PFTPs pending, the amount of cache used by the files in local cache, and the amount of cache reserved for the requests currently in the queue. In addition to these measurements by the CM, the QM keeps track of information related to the status of queries. Specifically, it keeps also track of the number of queries waiting to be served or being served, and also the amount of cache actively being used, i.e., cache used by files that are being currently processed by some query. In this context, a query is considered as being served if it is currently processing a file, or if it has a file in local cache to process.

All these quantities can be displayed dynamically when the system runs and can be used by the STACS administrator to tune the policies of the system to overcome bottlenecks. For example, one of the parameters that can be set dynamically is how much pre-fetching to perform on behalf of each query. If there is a lot of available disk cache, and the PFTP queue is small, one can increase the number of pre-fetches, so that queries have at least one additional file in cache as soon as they finish processing a file. An example of such measurements displayed for a particular run are shown in Figure 7.

Another reason for keeping track of these measurements performance, is to observe whether the system resources are "balanced", i.e. used well for a typical query mix. In particular, it is important to understand where the bottlenecks are, and if some resources (tape drives, disk cache, and network resources) are underutilized. Accordingly, this can be used as a guide for adding the right kind of resources to the system to achieve better system performance.

3.6 Recovery from Crashes

One of the very important, even if rarely used, features of the CM is the capability to recover from crashes and return to its state before the crash. By crash we mean a real crash of the CM, which although very unlikely (we have run the CM for weeks without a glitch) cannot be put aside, but also the situation where the machine where the CM runs needs to be rebooted. Given the fact that a set of queries can take days to process it's of utmost importance that the system can return to its state before a crash without the users having to relaunch all the queries again. The CM does this by logging to a "recovery" file the list of requests that were not served yet. Once a new request arrives, information about it (file id and query id) is logged to a file, and after a request is served (a file is transferred) the associated information is removed from the same file. If the CM happens to crash or the system where it runs needs to be shut down, the CM can easily return to its previous state by reading the "recovery" file, and checking if the files were correctly transferred and are currently in cache. For any files not correctly transferred or not transferred at all, the CM relaunches the logged requests.

4 Conclusions

We described in this paper a real implementation of a storage access queuing and monitoring system to be used in high energy physics applications. The system is practically ready to be deployed and has been in a testing phase for the last few months. The system has been tested against a 1.6 TB federated database of synthetic data stored in 170 tapes. We have demonstrated the value of such a system in insulating the user's application from the details of interacting with a mass storage system. Specifically, the system enables the user to submit a query of what is needed, and the system finds all the files that need to be read from tape, schedules their caching so that files can be shared by multiple users, minimizes tape mounts, handles transient errors of the mass storage system and the network, and monitors performance. Such a system is particularly valuable for long running tasks (many hours) of 100's of files, where restarting a job because of a failure is not a practical option. Future plans include the application of the system in distributed multi-site grid infrastructure. In this setup, there can be multiple sites that have mass storage systems, and each site may have a shared disk cache for its local users. We envision the Cache Manager's functions to be associated with each storage resource in the system. An open (and difficult) problem is how to coordinate these distributed resource managers in order to support multiple users at various sites in the most efficient way. We also plan to apply this technology to application areas other than high energy physics.

References

[1] D. Düllmann. Petabyte databases. In *Proceedings of the 1999 ACM SIGMOD*, page 506, Philadelphia, Pennsylvania, 1-3 June 1999. ACM Press.

[2] A. Hanushevsky. Pursuit of a scalable high performance multi-petabyte database. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems*, pages 169–175, San Diego, California, 15-18 March 1999. IEEE Computer Society.

[3] J. Shiers. Massive-scale data management using standards-based solutions. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems*, pages 1–10, San Diego, California, 15-18 March 1999. IEEE Computer Society.

[4] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 214–225, Cleveland, Ohio, 28-30 July 1999. IEEE Computer Society.

[5] A. Sim, H. Nordberg, L. M. Bernardo, A. Shoshani, and D. Rotem. Storage access coordination using CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 168–175, Edinburgh, UK, 5-6 Sept. 1999. IEEE Computer Society.