Real-time data access monitoring in distributed, multi-petabyte systems

Tofigh Azemoon, Jacek Becla, Andrew Hanushevsky and Massimiliano Turri

Stanford Linear Accelerator Center, 2575 Sand Hill Road, Menlo Park, CA 94025, USA

azemoon@slac.stanford.edu

Abstract. Petascale systems are in existence today and will become common in the next few years. Such systems are inevitably very complex, highly distributed and heterogeneous. Monitoring a petascale system in real-time and understanding its status at any given moment without impacting its performance is a highly intricate task. Common approaches and off-the-shelf tools are either unusable, do not scale, or severely impact the performance of the monitored servers. This paper describes unobtrusive monitoring software developed at Stanford Linear Accelerator Center (SLAC) for a highly distributed petascale production data set. The paper describes the employed solutions, the lessons learned, the problems still to be addressed, and explains how the system can be reused elsewhere

1. Introduction

As High Energy Physics (HEP) experiments grow in size and their data volume reach the multi-petabyte scale, the complexity of data storage and delivery systems grow accordingly. Large systems with thousands of users are often distributed geographically across different time zones, managed by different groups of administrators, and run on different operating systems. It is vital for efficient functioning of such systems to develop detailed understanding of how they perform and promptly respond to any arising issues. Monitoring such systems is certainly non-trivial. We know of no off-the-shelf tools that report detailed byte-level file activity for studying data access patterns. The two most popular distributed monitoring systems, Ganglia [1] and Nagios [2] that are designed to monitor large clusters provide either very general OS metric data or only offer service status monitoring.

SLAC [3] hosts the production data for BaBar [4] that is one of the largest HEP experiments currently on-line. BaBar uses xrootd [5] for its data access and with over 2 petabyte of data [6] is an ideal case for developing a large scale monitoring system. Xrootd is a highly scalable data server that includes load balancing among data servers and has transparent recovery from server crashes. These features together with automatic tape to disk staging of data have resulted in good system performance and system maintainability. But they do not help the system managers in their planning for system expansions to understand issues like typical data access patterns and how they change over a period of time, or identify the most active users at each site and the amount and type of data they access or the type of files most commonly accessed. To answer such questions, we extended xrootd by building a real-time, unobtrusive monitoring system capable of monitoring operations on highly distributed, multi-peta scale data sets. It is important to note that although the existing system was developed for the BaBar experiment it is an open source package with well documented client-server APIs, so it can be reused in other experiments assuming one can modify the server-side code.

2. Architecture of the Monitoring System

The monitoring system is capable of monitoring in real time any number of data servers distributed worldwide. Data servers send data using UDP packets. The main advantage of this is complete isolation of the mission-critical data servers from the monitoring system. On the downside, UDP does

not guarantee data delivery, so potentially some data might be lost. But due to the statistical nature of monitoring this is not an issue. Also, UDP packets can arrive in any order, a feature that needs to be carefully handled by the receiving end. Decoded data is stored in the form of ASCII files. An application loads the decoded data to a database and another application prepares the data for fast analysis, mostly by building summary tables that are used by the web application. The key components of the system are described below.

2.1. Xrootd

The xrootd server provides POSIX-like access to files and their enclosing directory namespace. Within xrootd the monitoring is implemented as part of the protocol plug-in and the client is given some limited access to the monitoring activities through the xrootd requests as discussed below. The Monitoring was designed with focus on four major issues:

2.1.1. Impact on client requests. The recording of the event must have almost no impact on the request itself to make the monitoring accurate and not impede client performance. This implied that the monitoring itself required very little resources (CPU, memory, I/O) and optimally did not serialize client requests in order to maintain low request/response latency. A low overhead design necessitated using compact binary data structures to avoid data translation overhead. Also precise timing of events can prove very costly. Hence, the design used statistical event recording where events are gathered in a precise time window but no particular event itself is given a precise time. This minimizes using high overhead processor timing facilities.

Events occur in some order across all clients and for monitoring purposes the timing order of all events and their relationship to each other should be preserved, that is, event recording should be serialized. However, the xrootd read/write path executes without obtaining serialization locks in order to achieve extremely low latency [7]. To avoid introducing serialization in xrootd each client is allowed to generate its own event stream instead of generating a single coherent monitoring stream. Hence, xrootd presents multiple independent monitoring streams and it is up to the data collector to serialize those to obtain an ordered list of events.

2.1.2. Robustness in the event of multi-mode failure. Fault tolerance is a strong feature of xrootd. When deployed in a clustered environment, xrootd can ensure access to data by re-hosting data from failed servers and redirecting clients in real time to the new data location. The monitoring system design needed to be equally fault tolerant in the sense that any failure in the monitoring path would not affect the server in any way. This was achieved by simply making sure that system resources used for monitoring were always bounded. The design specifically focused on data delivery mechanisms that used limited buffering and would preferentially lose data instead of slowing down processing in the event of a data flow backup.

2.1.3. Precision and specificity of collected data. We addressed monitoring specificity by allowing clients to inject contextual as well as statistical information into the monitoring stream. The injected information retains its relationship to the events that occur at the time the client generates additional information. This allows server events to be related to client events and substantially increases the precision of the monitoring stream. Additionally, clients are allowed to enable and disable monitoring of server events with respect to themselves.

2.1.4. Real time scalability. One of our aims was to have a real time data access monitoring that was scalable and at the same time could capture very detailed information for special studies. These two requirements were at odds with one another since a detailed monitoring stream could easily create a data flow that would saturate the data collector agent either because of CPU or I/O constraints leaving little resources to extract data for real-time purposes. We solved this problem by allowing multiple

data collectors for load sharing purposes, and by providing a means to separate low rate events and sending them to a different data collector. As a result, the monitoring can run in either light or bulk mode as depicted in figure 1. The light mode deals with session and file related data such as session start and stop times, host names, process ids, file open and close times, file paths and number of bytes read/written in each session as well as the xrootd restart time on each data server. It is used for real time displays. The bulk mode allows for much finer granularity by tracking each file access, recording its type (read/write), offset, and number of bytes accessed. The large volume of data and resulting load on the system makes this mode unsuitable for real time and sustained monitoring. It is meant to be used to collect a sample of the I/O stream during short periods of time for further off-line analysis to determine access pattern optimizations.
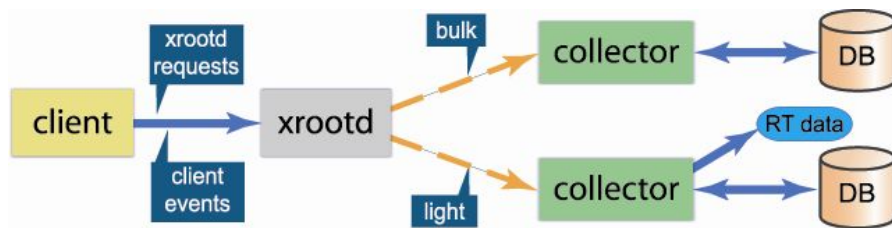

Figure 1. Monitoring architecture with 1 client/server and 2 collectors

2.2. Single Site Light Monitoring

In our experience, two dedicated hosts are sufficient to monitor a single site, one for monitoring and one for web server. For security reasons, it is advisable to separate web application from the monitoring internals. The monitoring server hosts the database and is designated to run the database applications. Figure 2 shows a typical configuration. The monitoring system is simply set up by configuring each monitored data server and initializing the monitoring and web servers. This involves initializing the database by running the *create* application, and starting three other applications: *collector/decoder, load* and *prepare*. Setting up the web server requires a few configuration changes on the Tomcat server side to allow connection to the database [8].
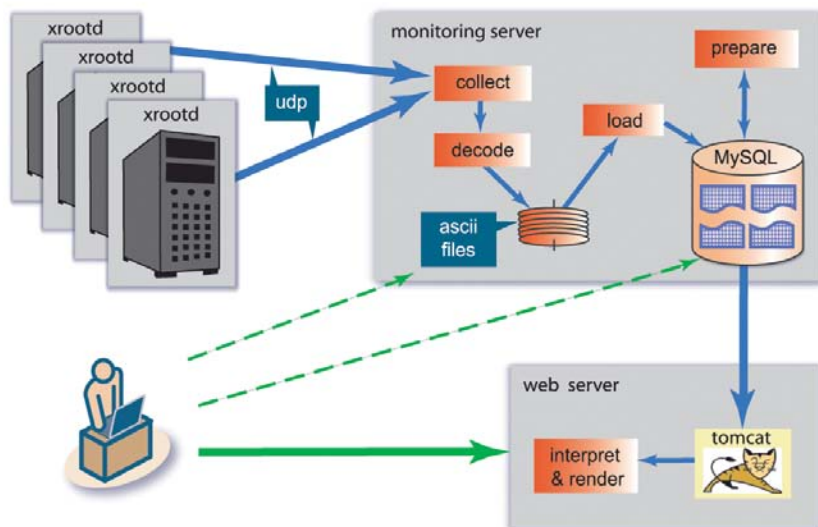

Figure 2. Typical light monitoring configuration for a single site

2.3. Multi-Site Light Monitoring

In this configuration the main site hosts the database server and receives the decoded monitor data from all other sites and loads them into the database. Each site runs its own collector to collect the data from local data servers. The decoded data, in ASCII format, is streamed to the main site over the web using a custom made package. The database tables for different sites are decoupled so they can be updated in parallel by running multiple instances of *load* and *prepare* to reduce latency. To provide ultimate scalability for very large systems with many sites and hundreds of servers, database update can be further parallelized, for example as shown in figure 3.
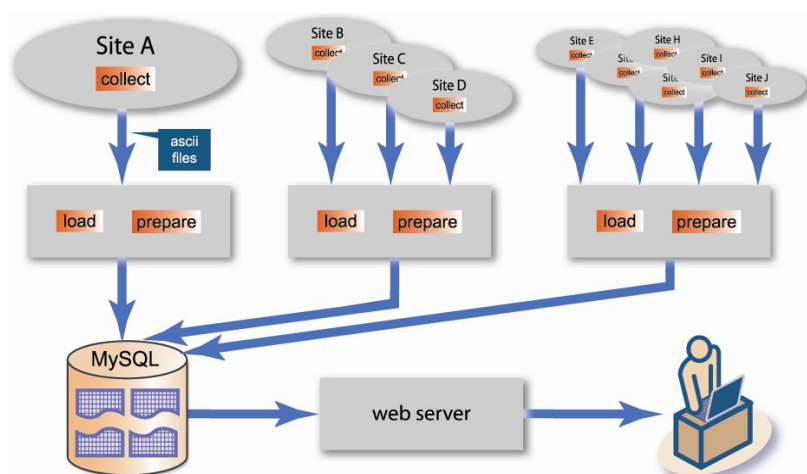

Figure 3. Example configuration for light multi-site monitoring

2.4. Real-Time Considerations

Most of the information on the data server side is flushed when a UDP packet is full or a specified time elapses (default 1 minute.) This data is decoded immediately and flushed to a log file in short intervals (typically 10 seconds.) Loading to the database introduces another short delay, related to the fact that *load* and *prepare* wake up every minute to process new data. So in practice, the information is available in the database for web browsing a few seconds to a few minutes after the event happens, depending on configuration. This delay can be shortened at the expense of a higher load on the system.

2.5. Generic Design

In designing the system, while our main aim was to provide monitoring for the BaBar data access via xrootd we have made special effort to make it generic enough that other groups can use all or part of the system. Most monitored parameters, such as user, client host, data server etc. have standard definitions. One area that needed special attention concerned file classification.

In a petascale dataset comprising millions of data files not all files are accessed with the same intensity. The files most frequently accessed are those relating to the on going analyses. To optimize the data access it is essential to be able to identify the files or classes of files that are in high demand in order to make them more readily accessible. The monitoring system follows the file classification used in the data production and for each file type it can monitor aggregate values such as number of files, total bytes read or number of users accessing them. It can also provide the list of files satisfying certain criteria for each class. Since file classification is designed for the purpose of identifying full datasets for all possible studies this kind of monitoring provides a constant polling of the user community for its on going requirements.

In general the datasets are classified according to many different criteria and a file may belong to several classes at the same time. The information needed for the classification is either embedded in the file name or is contained in look up tables or a database. Since each experiment has its own naming convention the system is built in a generic way where the class names are supplied in a configuration file. On the other hand no a priori knowledge of the class member names, i.e. file types, is required. This information is dynamically built up as the database is populated and an experiment specific code provides the corresponding values for each file. As an example files may be classified as *real* or *simulation* within the class *data-type* with the convention that these values appear between the second and third slash in the file path. The file class *data-type* is declared in the configuration file but the system has no knowledge of the values *real* and *simulation* until the files /root/real/... and /root/simulation/... are accessed for the first time.

2.6. System Expansion

The system in its present form allows easy expansion in two ways without major changes to the code. These are adding new sites to be monitored and new file classifications. A site is declared in the configuration file by giving its name, startup date and time zone. A separate application is used to upgrade the database by creating the site related tables. At the same time a simple expansion of the directory structure is needed on the database server to accommodate the new site.

Adding a new file classification not only requires creating new tables it also affects some of the existing ones. Therefore the database upgrade needs a special outage. The code that evaluates file types as well as the configuration file is updated to reflect the new classification before the database upgrade. The outage could be quite long if one is dealing with multi-million row tables. However, the information arriving from the xrootd servers during the outage is buffered and subsequently processed and the database integrity is respected.

3. Implementation Details

This chapter highlights implementation details of the key components of the monitoring system.

3.1. Xrootd

Monitoring is configured in xrootd by the "monitor" directive. This directive specifies the events to be monitored, event buffer size and flush interval, monitoring window size, and data stream destinations. The events that can be monitored include file-related requests (open and close file), I/O request (read and write), client specified data submitted using xrootd protocol, and client logins and disconnects. Each event is 128 bytes of highly encoded binary information in network byte order. This was determined by the desire to compress the information as much as possible using the least amount of CPU resources. The buffer is always prefixed by a header that specifies the event count, the buffer type, and a relative sequence number that is used to order buffers in the proper time sequence. Several types of buffers exist: file and I/O trace events, and dictionary entry for user/path combination, user/information combination, user login name, and user authentication information.

Typically, only one dictionary entry is placed in a buffer and flushed to ensure that the entries are received as early as possible as they map variable information (e.g. file path) to a unique number called a dictid. The dictid is then used in subsequent records in order to conserve space yet provide a tie to the external context in which the event occurred. The data collector is responsible for translating dictid's to the proper context. File and I/O events are collected in a buffer and flushed when the buffer is full or the flush time limit is reached. The buffer always starts and ends with a timing mark. Each mark consists of two UNIX timestamps that record the original time and the latest time the mark is generated. The need for two timestamps is explained below

In figure 4, each timing mark is logically placed in the buffer at a set interval corresponding to the configured window size. The events are recorded within a timing mark window defined by a start and an end mark. When two marks are adjacent to each other, indicating zero events in that window, xrootd merges them together. To maintain the notion of a window, two times must now be maintained as shown on the left hand side. The first time closes off the previous window while the second time starts the next window. A timing mark mechanism provides flexibility in controlling the tradeoff between accurate event timing versus the cost of doing so. Generally, a 60 second window size is sufficient to properly discriminate the overall order of events. When a buffer is flushed, it is written to a UDP based socket whose end-point is specified in the configuration file.
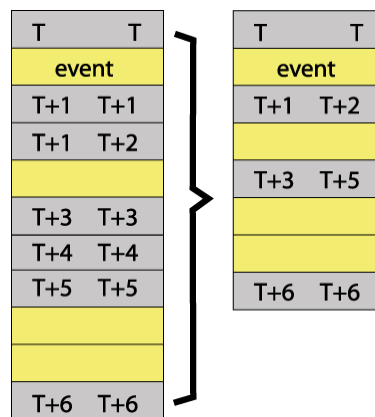
Figure 4. Timing marks in event

3.2. Collector
For performance reasons the collector is written in C++. It merges all incoming data from multiple data servers into a uniform coherent time stream. Care is taken to appropriately order the UDP packets using the time-stamp and the sequence number assigned to each packet. The latter also helps to discover if any packets are lost due to network congestion. The collector uses multi-threading to efficiently decode data from multiple data servers. Based on our experience at SLAC, a single collector should be able to handle several hundred servers configured to run in light mode. The performance with bulk mode heavily depends on underlying hardware used, in particular disk IO speed. At SLAC, we were easily able to handle 50 data servers in bulk mode on a modest 2-CPU host.

3.3. Database and database applications
The most natural way to manage data from light-mode monitoring is through a relational database. For performance reasons we have chosen MySQL. The applications that create and maintain the database are written in Perl and access the database via Perl DBI module and are discussed below.

3.3.1. Create. This application creates and initializes the database. It chooses the optimum values for creating tables using the information contained in the configuration file.

3.3.2. Load. This application is responsible for loading the collector data into the database. In the current configuration it wakes up every minute to transfer the input data to its work area. During this short transfer time it places a lock on the input file to prevent concurrent collector updates. The load on this application is kept to a minimum to maintain low database latency and a built in recovery mechanism prevents data loss due to unplanned program terminations.

3.3.3. Prepare. This application is designed to maintain up-to-date summary and statistics tables used by the web application. It provides historic tables of varying granularity and time span, ranging from

one hour to full length of time the monitor data is available, that track the time evolution of number of users, jobs, files and bytes transferred. At fixed intervals each historic table is updated, preferentially using entries from a table of shorter time span but higher granularity. The "last hour" table, however, is updated every minute using the actual values from detailed tables. When *prepare* is restarted after a period of inactivity or is started after running *reload* application there can be missing data points that can not be determined in above manner. In this case we resort to the detailed tables and use a special algorithm to get equivalent results.

3.3.4. Upgrade. This application is used to add a new site or file classification to the database. Adding a new site is non-intrusive. If the new site is not included in the instances of *load* and *prepare* that are running, it suffices to start another instance of each application for the new site. Otherwise the corresponding applications are restarted with the new configuration file. In case of adding a new file classification a database outage is needed since some of the existing tables are also affected.

3.3.5. Reload. It is important to have the possibility of recreating the database at any moment not only as a safeguard against losing valuable data but also as a means of implementing improvements or major schema changes. *Reload* has the capability of partially or fully reloading the backed up ASCII data to the database. It simulates the real time loading by breaking the data into one minute intervals. Even during a full reload it reuses as much of the existing tables as possible such as the ones with file and user information. This enhances the reload speed by avoiding resource intensive operations such as redoing file classifications.

Certain combinations of database applications can not run in parallel. For example, when *load* is restarted after a period of inactivity it may need to catch up with a large backlog of input data. During this time the database is not fully up to date and *prepare* must wait until the backlog is cleared. Or *load* and *prepare* can not run during a schema upgrade for a new file classification. The communication among database applications is established through semaphore files. Each application is started and stopped using a special script that among other things is responsible for graceful termination of the application and handling of the semaphore files.

3.4. Web Application
This application is written in JSP3 which offers a fast and easy way to create dynamic web applications and provides a set of tags to access databases using JDBC. These features were essential for creating a web front end whose pages and content is dynamically driven by database tables. The summary plots are created dynamically on the server side using the AIDATLD tags. These are JSP compatible tags that allow one to easily generate scientific plots from the outcome of database queries. To display tabulated data we used the Display Tag tags. With this technology it was possible to efficiently couple database queries with html tables with variable number of rows and sortable columns. Maven 1 is the technology we use to manage and build the web application. The web application is distributed as a war4 file and works in Tomcat 5.5 (or greater) servers installed on any OS.

3.5. Time Zones
The UDP packets arriving from different sites contain local time information leading to a mixture of time zones when viewed collectively. For the sake of clarity and uniformity the data from all sites must be presented coherently by the web application using a single time zone. Storing the timestamps in a unique time zone in the database inevitably leads to simpler, hence faster queries. For this reason the timestamps are converted to GMT at the decoding stage. The web application, however, offers a pull down menu to select the viewing time zone. At present the choice is restricted to the time zones corresponding to the contributing sites but could easily be expanded to cover additional sites.

3.6. Job Definition

The job attribute encompasses all operations performed by a user under one process id on one client node. This is a very useful parameter to monitor as it can be used to identify analysis or production related issues and bottlenecks. For monitoring purposes we are only concerned with the data access part of the job and not the user code. A job can span many parallel and consecutive sessions whereby the client host connects to data servers to perform file read and write. In this context a job starts with the first session the client node connects to a data server and ends with the last disconnect from a data server. A job is assumed to be running if it has at least one open session at that time. It may happen that for a short time there are no open sessions before the next session starts. During this time the job is dormant. As the monitoring system has no a priori knowledge of the last session within a job one must predefine the length of time a job can be dormant before it is declared as finished. In principle this can be as long as the cycle time of the process id on the client. However, one should choose the smallest possible value so that the database reflects the most accurate picture of the job status at any time. By definition the session related data should be sufficient to determine the status and duration of a job. But occasionally some session information may get lost due to xrootd restarts or system crashes. In this case the xrootd restart time or file related information is used to determine approximate timings.

3.7. Backup, Outages, Journaling

We have taken great care that the monitoring data arriving from the xrootd servers remains constantly available for reprocessing in its entirety. The only time that a small amount of information is lost is on the rare occasions when the collector is stopped and restarted during upgrades or server restarts. At every load cycle the decoder output is moved to a journal directory for processing and is immediately appended to the latest backup file which is closed at regular and configurable intervals. During outages for database maintenance or upgrades the decoder keeps on updating its output file and the backlog is processed after the outage. Detailed journal files are maintained for recovery purposes in the event that an application stops abruptly.

3.8. Configurability

A single configuration file that is used by all applications serves to set up the database and control its operations. The only hard coded numbers in the system are the one minute load cycle and the bin sizes in the statistics tables that are used for time plots. To set up the directory structure where the collector output and the journal and backup files reside only the base directory and the site names must be declared in the configuration file. All control parameters have default values that can be over-ridden in the configuration file.

4. BaBar Production System

BaBar has so far collected over 2 PB of data. Data is processed and served from centers located in USA, Canada, France, Italy, Germany and UK, and accessed by more than 600 users from 80 institutions worldwide. All this is centrally monitored at SLAC. Worldwide, BaBar has access to hundreds of data servers with nearly 1 PB of disk space and to several thousand processing nodes that access data from a pool of well over one million files.

4.1. File classification in BaBar

For monitoring purposes we have implemented two of the file classifications used in BaBar and will be adding more in future. The first class is *data-type* which indicates whether the data is real or simulated and in either case whether it is filtered or not. In BaBar filtering is referred to as skimming. The second class is *skim* which contains the filter names. Each skim refers to a set of selection criteria that is used to obtain the corresponding data set. As the experiment progresses more skim names are added to the set.

4.2. Monitored Parameters

The distribution comes with a web application that renders the contents of the database as plots and tables for all participating sites. Information is presented in a hierarchical structure allowing easy navigation from summaries to detailed pages following links in the tables or using the navigation menu. For example one can monitor the number of jobs, users and open files, or view tables of the most active users, most used files and file types and obtain detailed information for individual cases. Two examples for SLAC are shown in figures 5 and 6. A drop down menu selects the time period that can be last hour, day, week, month or year. In case of the top performers table, shown in figure 5, the number of rows for each section is also selectable up to a configurable maximum. We also provide a repertoire of popular queries to be continuously expanded through user feedback.

Table rows: [5 ▾]   Time Period: [Last Hour ▾]   Site: [SLAC ▾]   [Update]

Top active users

User Name	Now			Last Hour			
	Number of Jobs ↑	Number of Files	File Size [MB]	Number of Jobs	Number of Files	File Size [MB]	MB Read
ayarritu	615	139	65,987	430	146	65,802	41,360
jregens	360	405	371,874	64	317	303,252	143,852
cschill	281	32	27,133	79	30	25,301	4,892
feltresi	149	106	167,528	70	143	218,873	74,552
torsten	72	99	83,673	184	1,532	630,092	235,327

Hottest dataTypes

dataType Name	Now				Last Hour				
	Number of Jobs ↑	Number of Files	File Size [MB]	Number of Users	Number of Jobs	Number of Files	File Size [MB]	Number of Users	MB Read
SPskims	998	739	632,651	11	663	340	304,938	6	120,728
SP	652	1,839	1,961,610	12	981	506	474,819	7	159,512
PRskims	93	650	811,152	7	204	83	107,807	2	62,265
PR	66	600	453,640	6	265	1,454	525,498	3	174,754
cfg	0	0	0	0	8	1	7	1	10

Hottest skims

skim Name	Now				Last Hour				
	Number of Jobs ↑	Number of Files	File Size [MB]	Number of Users	Number of Jobs	Number of Files	File Size [MB]	Number of Users	MB Read
BtoRhoGamma	591	139	65,987	1	458	146	65,802	1	41,360
DstToD0PiToVGamma	262	86	33,138	1	70	41	16,171	1	4,668
BToDlnu	115	118	186,026	2	125	145	222,200	2	74,568
AllEvents	76	394	508,309	3	210	84	108,365	3	62,268
Tau11	4	95	130,103	1	3	6	149	0	127

Hottest files

File Path	File Size [MB]	Now Number of Jobs ↑	Last Hour Number of Jobs	MB Read
/store/PRskims/R18/18.6.3d/AllEvents/00/AllEvents_20006.04HB.root	1,690	2	15	1,630
/store/PRskims/R18/18.6.3e/AllEvents/05/AllEvents_20502.04HB.root	1,688	1	17	1,636
/store/PRskims/R18/18.6.3e/AllEvents/05/AllEvents_20502.01.root	1,689	1	17	1,635
/store/PRskims/R18/18.6.3e/AllEvents/05/AllEvents_20500.03HB.root	1,688	1	19	1,641
/store/PRskims/R18/18.6.3e/AllEvents/05/AllEvents_20500.01.root	1,689	1	19	1,640

Figure 5. Example summary page showing "Top Performers"

4.3. Statistics.

We started deploying the monitoring system at SLAC in mid 2005 and gradually all Tier A sites, six in all, started participating. Currently, the database contains 200, mostly multi-million row, tables and exceeds 40 GB in size. The largest table has over 130 million rows. In the last two years over 30 million jobs were recorded, which were submitted by more than 600 different users. These jobs ranged

from few minutes to many weeks in duration and accessed over 1.3 million distinct files a total of 143 million times. Currently over 200 skims are monitored both for real and simulated data. During peaks of activity we have observed over 100 active users for prolonged periods of time processing many thousand jobs and accessing well over one hundred thousand files at any given moment.

Now		Last Hour	
Number of Running Jobs	203	Number of Finished Jobs	831
		Total Duration of all Jobs [DAY HH:MM:SS]	74 16:46:57
Number of Open Sessions	388	Number of Closed Sessions	1,865
Number of Open Files	146	Number of Accessed files	1,241
		Volume of Data Read [MB]	719,109
		Volume of Data Written [MB]	0
Number of Client Hosts in Use	157	Number of Client Hosts Used	593
Number of Server Hosts in Use	44	Number of Server Hosts Used	50

Figure 6. Example of monitoring information for one user

5. Future Plans

5.1. Monitoring tape to disk staging

In multi-petabyte experiments it is neither economical nor practicable to store the entire dataset on disk. Disks are much more expensive than tapes for the same data volume and more importantly they consume power even in the absence of IO. Since only a fraction of the data is accessed at a given time the files are continuously staged and purged. Monitoring staging would be very valuable in detecting the inefficiencies in the automatic staging and in planning disk upgrades for optimum data access.

5.2. Multi-experiment monitoring

In a data grid world where computing resources are used jointly by several experiments it is highly desirable to have per experiment view of the shared resource usage. The work is in progress to introduce an experiment layer in the monitoring system.

5.3. Fractional data usage

Filtering data is a resource intensive process and its usage should be carefully monitored to avoid producing types of data that are no longer in demand. In BaBar an off line monitoring system uses the production database and the xrootd log files to determine the fraction of data accessed for each skim type. We are in the process of setting up a separate stream that will update the monitoring database with the amount of data produced. This is being done in a generic manner so that it can be used as part of the monitoring system by any experiment.

6. Summary

At SLAC we have developed an unobtrusive data access monitoring system for petascale datasets. The monitoring system is decoupled from the underlying data access server and its scalable architecture allows it to be used for monitoring distributed systems of virtually any size. We have deployed it in the last two years to monitor the data access in BaBar experiment that has over 2 petabyte of production data. The system can either be used to monitor the access in real time in lightweight mode, or as a tool for periodically collecting fine-grained access statistics for further offline analysis.

References

[1]   http://ganglia.sourceforge.net/
[2]   http://nagios.org/
[3]   http://www.slac.stanford.edu/
[4]   http://www-public.slac.stanford.edu/babar/
        Aubert B *et al* 2003 BaBar Detector *Nucl.Instrum.Meth* . A **479** 1-116
[5]   Dorigo A, Elmer P, Furano F and Hanushevsky A 2005 Xrootd - a highly scalable architecture
        for data access *The 7th WSEAS International Conference on Automatic Control Modelling
        and Simulation (Prague, Czech Republic, March 2005)* available on-line at
        http://xrootd.slac.stanford.edu/
        Boeheim C, Hanushevsky A, Leith D, Melen R, Mount R, Pulliam T and Weeks B 2005
        Scalla: scalable cluster architecture for low latency access using xrootd and oldb servers
        http://xrootd.slac.stanford.edu/papers/Scalla-Intro.htm
[6]   Becla J and Wang D 2005 Lessons learned from managing a petabyte *CIDR Conf. (Asilomar,
        Ca, USA, January 2005)* http://www-db.cs.wisc.edu/cidr/cidr2005/papers/P06.pdf
[7]   Weeks B and Hanushevsky A 2005 Xrootd performance
        http://xrootd.slac.stanford.edu/presentations/Xrootd_Performance.htm
[8]   http://xrootd.slac.stanford.edu/examples/monitoring/index.html