DZERO DATA AQUISTIION MONITORING AND HISTORY GATHERING

G. Watts, University of Washington, Box 351560, Seattle, WA, 98195

Abstract

The DZERO Collider Experiment logs much of its Data Acquisition Monitoring Information in long term storage. This information is most frequently used to understand shift history and efficiency. Approximately 16 kilobytes of information is stored every 15 seconds. We describe this system and the web interface provided. The current system is distributed, running on Linux for the back end and Windows for the web interface front end and data logging. We also discuss the development path we have taken for the database backend, from use of root, to Oracle, and back to root, and the reasons for the change in design.

INTRODUCTION

The DØ Detector, located at the Tevatron, underwent a major upgrade before the start of Run 2, which commenced in March 2001. One component of this upgrade is the Level 3 Trigger/DAQ system, whose capacity was increased to handle higher data rates and larger event sizes. The software DAQ Monitor system received a complete rewrite for this upgrade. The new system is distributed, makes heavy use of XML for communication, is extensible, and supports reporting data at less than one second intervals. This paper describes an archive application used to log a subset of available monitor data in perpetuity. The application is called *l3xHistoryViewer*.

The l3xHistoryViewer application collects data at 15 second intervals and stores the data. A web interface can then extract the data and generate arbitrary plots. Since this project started 2.5 years ago, the data store has been redesigned twice. Its first incarnation used ROOT as a backend to store the data, the second Oracle, and the third ROOT. After describing the system in more detail, this paper will examine some of the reasons for the changes in design.

THE LEVEL 3 TRIGGER/DAQ AND MONTIORING SYSTEM

The Run 2 DØ Trigger system uses a multilayer trigger system typical of a large collider experiment. Level 1 is a hardware based, dead timeless trigger capable of making fairly simple decision. Level 2 is a combination of custom hardware and Single Board Computers, and finally Level 3 is a farm of general purpose, commodity, PCs running trigger software written by physicists, in C++.

Level 3 Trigger/DAQ is based around commodity network components and PC's. *Single Board Computers* (SBCs) read data from VME based *Read Out Crates* (ROCs). The SBC's send the data cross the network, via a large Cisko network switch, to a PC for event building and filtering. A *Routing Master* (RM), connected directly to the Level 1/Level 2 trigger framework hardware steers each event to a free, properly configured, farm PC. A *Supervisor* interacts with the DØ online run control (COOR) and configures the components of the trigger and DAQ system as requested [1].

The system contains a large number of components that must be monitored to assure smooth running. There are 63 ROCs and their SBCs and 114 Level 3 Trigger Farm PCs, a RM, and a Supervisor. These are called *clients* in the monitoring system. Many of these have more than a single program that produces monitor information, and there are a number of other sources of monitor information throughout the online system.

The monitor system design is based around a central *monitor server*, as shown in Figure 1 [2]. Clients, sources of data, and displays, which request data, connect to the monitor server. The displays request data from the server, and the server queries the clients and returns the information. The display's request can be for data from many different machines, and the monitor server will collate the data from all the clients into a single reply. The client is never queried for information unless a display requests that information. This is designed to minimize the load on the DAQ components and also provide for the possibility of complex, expensive monitor information for debugging; it is only generated when requested.

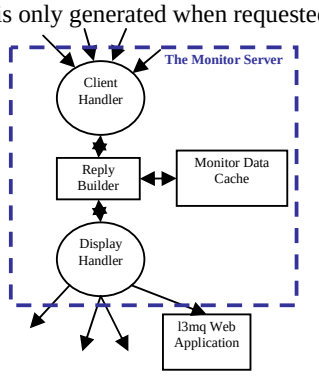

Figure 1: Block diagram of the DØ DAQ Monitor System. Clients connect to the Client Handler, and displays to the Display handler. The Reply Builder uses the Monitor Data cache to satisfy as many requests as possible from the displays, and for others uses the Client Handler to make a further request. The l3mq Web Application caches complex queries and exposes them as a simple web query.

The monitor server caches the most recent information from each client. A display can specify a *staleness* parameter. The monitor server will only re-query a client

if the display's request requires more recent data. Most displays ask for data one second or less in age. Typically quite a few of these displays are running and thus the cache is hit ~40% of the time.

All monitor server communication with both displays and clients is done over TCP/IP and in XML. We use Xerces [3] for much of our display side parsing, and wrote a simple light-weight parser for the monitor server. We use the standard C++ *std::string* and std::*ostringstream* objects to build replies in the clients. The std::string requires special care: the result must be preallocated or a large performance hit is taken reallocating and copying the string as it grows. Figure 2 shows an example of an XML message reply from the monitor server to a display.

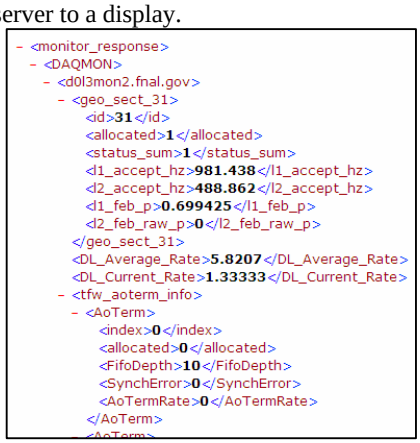

Figure 2: Sample XML response from the Monitor Server to a Display. The DAQMON tag is the monitor data type, d0l3mon2.fnal.gov specifies the machine the monitor information originates from, and DL_Average_Rate or geo_sect_31 is the actual monitor item name. The numbers are the data.

Typical client reply message sizes are of order 10 KB, and one of our larger displays can receive upwards of 100 KB worth of information. We have observed no performance issues other than forgetting to pre-allocate *std::string* (or not using *std::ostringstream*) and the overuse of XML to encode 200 connection states. Both easily fixed once the problem was identified.

The online system is protected by a firewall. A single external machine is allowed to access the internal monitor data. There are two ways for other external machines to access this data. First, there is a TCP/IP relay. The format of the incoming external monitor request is identical to the request normally sent directly to the Monitor Server. The format is checked carefully before the request forwarded to the monitor server. The second method is used by a number of clients, including the *l3xHistoryViewer*. A web application caches a particular monitor request and associates a name with that request (see Figure 1). A simple URL can be used to then request the monitor response. The monitor items returned by the request can be changed by a user using a web interface. In the case of l3xHistoryViewer this is particularly convenient as it allows one to alter the items archived using only the web interface. The web application is called *l3mq* (Level 3 Monitor request).

THE L3XHISTORYVIEWER

The logical design of l3xHistoryViewer is shown in Figure 3. Every 15 seconds l3xHistoryViewer backend requests a list of monitor values from the l3mq application. The returned XML is parsed. The data and a unique name are extracted for each monitor item. These items are then stored in a data store (ROOT or Oracle). A web application then reads the data at a users request and uses ROOT to plot the data and display it on the web.

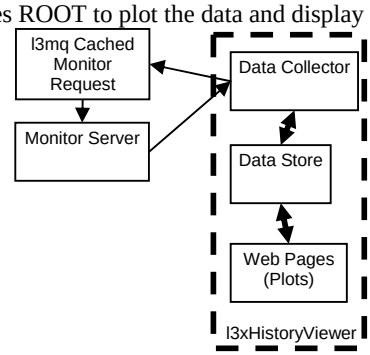

Figure 3: The logical design of the l3xHistoryViewer application. The data collector uses the l3mq monitor query cache to request the monitor data to be archived, and writes it to the data store. The web front end reads the data store and generates plots.

The backend, responsible for storing the data, has a fairly unique set of problems to solve. Most of these are driven by the fact that the monitor data is not stable over time. For example one monitor item requested is the event rate in each Level 3 farm node, which means a monitor item is sent for each of the 114 farm nodes. Unfortunately, these nodes are not stable and experience hardware related failures (one node a week tends to be offline for more than an hour). Further, a user may change the requested monitor items at any time using the l3mq web interface. Finally, the monitor server does not guarantee a response from every monitor source on every request. Sometimes a network connection is lost or a client machine is too busy to respond within the monitor server's one second timeout. The backend must also be able to write the data out to the backend store with low latency so a user can see up-to-date trends and plots from the web.

The web front end presents the user with a simple interface and caches common plot requests so they may be requested quickly using a single URL. The ROOT code that generates the plots can be altered by a user after appropriate authorization. The front end is also capable of calculating aggregate quantities – the average number of events processed by each active farm node, for example.

Monitor data is collected once very 15 seconds. The average number of monitor items is currently about 4000. If each item uses 4 bytes, then each hour this is 3.7 MB and about 31 Gigabytes per year.

THE THREE VERSIONS OF L3XHISTORYVIEWER

The design of the l3xHistoryViewer went through three iterations. The first was based on a ROOT data store, the second was Oracle based, and the third returned to a ROOT based data store.

All three versions of l3xHistoryViewer were written on Windows. The first version was written in C++, and the second in C#, and the third in a combination of C# and C++. All used ASP.NET to implement the web application and the web service for the front end. Most of the design of l3xHistoryViewer is driven by the backend data store. The application code could easily be ported to Linux using C++, Java, or, perhaps, Mono, and PHP or J2EE for the web front-ends.

ROOT I

This version of l3xHistoryViewer was a prototype to investigate the expected and discover the unexpected problems. Its design will only briefly discussed. The initial design was based on a previously existing program that dumped arbitrary monitor data to the screen for debugging. The XML parsing, hand coded, was not robust. The scheme to generate a unique name for each monitor item was not robust. The data was stored in a large root TTree; a single leaf was used for each variable, so the tree had as many leaves as there were monitor items (in excess of 4000). It is not possible to add a new leaf to a TTree once any data has been saved to the TTree, nor was it possible to mark a particular monitor item as missing if it temporarily dropped out. The result was that each time the list of monitor items changed the old TTree was closed and a new one created (one tree was stored per file). The large number of files and leaves proved to be a performance issue in version 3 of ROOT. The CPU time required to load a TTree with a large number of leaves is large; a single request for a plot often required 20 or 30 files and took more than 30 seconds. This issue has been substantially mitigated in version 4 of ROOT. Finally, the plotting code was tightly tied to ROOT, physically passing the TTree object around.

Oracle

Several lessons were taken from the first version of l3xHistoryViewer for this version: make the data format more robust against changing monitor items; remove the linkage between data store storage format and the plotting code; and add flexibility to the access of the data: plots by Tevatron Store Number, Run Number, Date, and other unforeseen things.

About the same time we were considering this redesign the DØ online system installed a beta-version of Oracle 8.1.7 on Linux and offered to host the l3xHistoryViewer data store as an Oracle database. This was attractive as management functions, like backup, would be taken care of by online system management personnel. This was even more attractive as we'd just lost several months of data due to a disk crash. The services of a Database Expert were also offered as most of us were fairly inexperienced in database development. Oracle ran on a dual 2.4 GHz P4 Xeon with about 100 GB of disk space, which wasn't configured as a RAID array.

The design follows Figure 3. After several iterations of database design we settled on the tables and relations shown in Figure 4. The layout was a compromise between speed and space. The main table, EVENT_INDEX, contains a single entry for each time slice (every 15 seconds). It's linked by its primary ID to all the values for that time slice in the EVENT_TO_VALUE table. Because many values read back constant, each possible value is stored once in the ITEM_VALUE table (for example, event rate will be zero for long periods of time between Tevatron stores). This compresses the amount of data stored. For further space savings, the ITEM_VALUE data are rounded to the .1% level. Finally, there is a link from each value to the name of the monitor item.

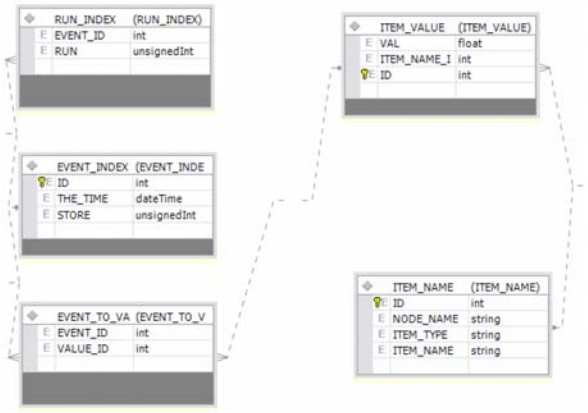

Figure 4: The layout of the Oracle Database for storing monitor data. The primary table, EVENT_INDEX, contains one entry for each time slice (4 per minute). The EVENT_TO_VALUE table contains one entry for each monitor item for each time slice. The ITEM_VALUE table contains all values of each monitor item, and is linked to the ITEM_NAME for the full ASCII item name.

In order to keep insertion time below ten seconds, all updates were done using the Oracle batch feature and using a stored procedure to maintain database consistency. Without these optimizations insertion was over 25 seconds.

Performance issues related to the extraction of the data were never fully resolved. The tests were performed after the data collection had run for 3 weeks. The query was keyed by a single run and monitor item request which should return 147 item entries. A straight forward, monolithic SQL statement with no database optimization took 247 seconds. A non-existent run took over 30 seconds to return a null result. The freeware tool Toad was used to examine Oracle's SQL plan and optimize the database [4]. The null result was returned in 0.3 seconds, and the 147 item request in 135 seconds. By extracting the ITEM_VALUE and ITEM_NAME and EVENT_INDEX locally to the front end web application, and doing the database JOIN there, speed was further

improved to about 30 seconds. Further progress was blocked due to a month of downtime and Oracle database corruption that was eventually traced to a bad disk.

Space was also an issue. After three months of data collection the database was projected to grow to over 100 GB for a full year's data collection. This was due to a combination of rollback logs, redo logs, and internal backups in the Oracle database. Some tuning was done to decrease this, but the effect was minimal.

ROOT II

In light of the Oracle problems we decided to re-write the data access layer in a third version. The data store would be again ROOT based, but in order to keep the flexibility a database would be used to store index data.

The database, currently hosted in Access, contains a table that associates runs and stores with dates. The root files are then archived in the file system, with directories named by date.

Figure 5 shows the layout of the two TTree's present in each root file. This is a compromise between the root file design of the first ROOT implementation and the complete flexibility of the Oracle database. The two trees are designed to prevent the addition of a new item causing the file to open and close and also minimize the number of branches in the TTree.

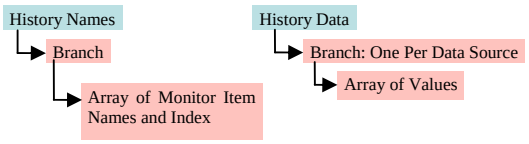

Figure 5: Contents of the two TTree's in each root file. The first tree, History Names, acts as an index into the second. The second stores only the monitor item data, in an array. Each branch of the second tree represents data from a single monitor data source.

The first tree, the HistoryNames tree, is an index. It contains an entry for each monitor item stored in the second tree, the HistoryData tree. The second tree stores the monitor data as a set of arrays. Each branch in the tree contains one floating point array. Each branch is named for the source of the data (for example, DAQMON). The index into the array for a particular monitor item is stored in the HistoryNames array. There is also a bool value stored in the HistoryData arrays to indicate the data is valid, allowing for the temporary disappearance of monitor items.

A new file must be written each time a new monitor type appears in the monitor stream: it isn't possible to easily add a branch to an already written tree. The system is designed to not forget monitor types that have disappeared, so if they return they will not trigger a new file. Because the values for those monitor items are zero ROOT should compress them away to almost nothing.

Despite the layout working against ROOT's compression software, two months of data indicates accumulating about 15.5 GB per year, which is easily manageable.

Extracting the data requires looping over each file that contains the required data. For each file, the HistoryNames tree is used to find the index into the HistoryData tree, and the data is then extracted and plotted. The plotter application can read the TTree as they are being written by the backend. The backend is tuned to update the ROOT file every several minutes so the plotter can extract the most recent information. Significant tuning of ROOT buffer sizes was required to assure that the in-memory foot print of the backend remained reasonable (currently requires about 40MB, 22MB of which can be traced to ROOT).

Performance tests were done on a PIII M 1.3 GHz CPU (a laptop). Extraction of ~900 items from 20 different files took 5.2 seconds on average. This performance was achieved only after a memory leak bug in ROOT was found and fixed. On the production machine, which is almost a factor of two faster, we expect the speed to improve further.

The plotting front-end has not yet been fully redeveloped for this version of ROOT. This will be completed shortly now that outstanding issues having to do with ROOT extraction have been addressed.

CONCLUSIONS

This project was initially thought to be a one-off taking of order 2-3 months to develop and get into production. Besides minor tuning and improvements, it was thought that very little day-to-day maintenance would be required.

Even the modest amount of data and the required flexibility required careful design. The use of the database was rolled back; in the end only data that is to be indexed on is stored in the database. The rest is stored external to the database. This keeps the database small and fast. Indeed, a low end database like Access has proved to be more than capable for holding the indices up to now. There is a further issue concerning development: Oracle is not a single person database. This makes it difficult to develop with as frequently database manager personnel must be accessible to make progress.

Starting with the lessons outlined in this paper it should be possible to put together a simple system similar to this fairly quickly. The tuning for speed, disk space, and memory management isn't easy, but there does exist a sweet spot.

REFERENCES

[1] *The DZERO Run II Level 3 Trigger and Data Acquisition System* (#477), Presented by D. Chapin at CHEP'04.

[2] DZERO Online Monitoring and Automatic Recovery (THGT004), Presented by G. Watts at CHEP'03. http://arxiv.org/abs/physics/0306195/

[3] The Apache XML Project, http://xml.apache.org/

[4] Toadsoft Home Page: http://www.toadsoft.com/