IMPROVING DATA RETRIEVAL RATES USING REMOTE DATA SERVERS*

Ted D'Ottavio, Bartosz Frak, Seth Nemesure and John Morris, Brookhaven National Laboratory, Upton, NY, U.S.A.

Abstract

The power and scope of modern Control Systems has led to an increased amount of data being collected and stored, including data collected at high (kHz) frequencies. One consequence is that users now routinely make data requests that can cause gigabytes of data to be read and displayed. Given that a users' patience can be measured in seconds, this can be quite a technical challenge. This paper explores one possible solution to this problem - the creation of remote data servers whose performance is optimized to handle context-sensitive data requests. Methods for increasing data delivery performance include the use of high speed network connections between the stored data and the data servers, smart caching of frequently used data, and the culling of data delivered as determined by the context of the data request. This paper describes decisions made when constructing these servers and compares data retrieval performance by clients that use or do not use an intermediate data server.

INTRODUCTION

Can a system be constructed that can read and display tens or hundreds of millions of stored data points to a user in a reasonable period of time (< 10 seconds)? What are the limitations and how can they be addressed? These are the questions that led to the research being reported on here.

The Controls logging system within the Collider-Accelerator Department at Brookhaven National Laboratory collects and stores measurement and setting data for offline analysis. The logging system, in place for about 10 years, uses a combination of user-generated and system-generated requests. These requests are passed to dedicated logger processes, which store the data to disk along with database records indicating how to map the requests to the stored data files. Data is retrieved and displayed by specialized software that allows a user to see what data was logged and to select data to be displayed.

The BNL logging system has become increasingly popular over the years, with the amount of data collected growing by a factor of 10 in the last 5 years, and expected to grow by another factor of 10 over the next 3 years (12 TB of time-series data was stored last year). Much of this increase is coming from requests to store data collected at high frequencies (720Hz to 10 kHz). In rough terms, about 500 MB of data is stored per day for every 1 kHz parameter requested. Retrieving and displaying a days worth of data for just a couple of these parameters means processing about a GB of data contained within files many times that size. For thick client applications that get data directly from the file system, this means transferring all of the data over the network to the client applications, picking out the requested data, and displaying it to the user before his or her patience has been exhausted. This puts a severe strain on both the network (to read the data) and the plotting software (to display it).

The solution reported in this paper uses a data server - a combination hardware/software solution. Its job is to handle user requests for logged data in the most efficient way possible. Its efficiency is a result of two key ideas:

- Read the data quickly – improve network speed to stored data, parallelize reads, and cache data.
- Cull the returned data - send the user only the data that can reasonably be distinguished on the display.

The use of an intermediate data server here follows similar efforts in place within the EPICS community [1] and at the CERN/LHC [2].

SYSTEM DESCRIPTION

The term data server denotes a collection of enterprise grade middleware applications deployed in a virtual cluster of dedicated and adopted process servers. At the core of the system is a single purpose application, whose role is to handle client requests for stored data. These requests are divided into discrete tasks, which are processed in parallel either locally or, in the event the core server cannot complete assigned workload, on one or more satellite servers. Result fragments from all tasks belonging to the same request, are reassembled by the core server and delivered back to the client.

Software Architecture Overview

Java Enterprise Edition 6 (Java EE6) was chosen as a base platform for the entire system. Benefits include a wide array of available web technologies [3] and a scalable business logic platform as well as built-in management features. Glassfish 3.1, which provides a complete open source Java EE6 reference implementation [4], was selected over other EE6 compliant applications servers.

The enterprise applications used to construct the data server are divided into three logical modules (Figure 1):

- Web component, which exposes a RESTful web service API to the remote clients.
- Request dispatch and assembly (RDA) Enterprise Java Beans (EJB) business module responsible for database communication, task scheduling and construction of the response objects.
- Request extract and transform (RET) EJB business module responsible for the data collection, caching and transformation. This component is divided

*Work supported by Brookhaven Science Associates, LLC under contract no. DE-AC02-98CH10886 with the U.S. Department of Energy

furthermore into local and remote sub-modules - the latter deployed on the satellite servers.

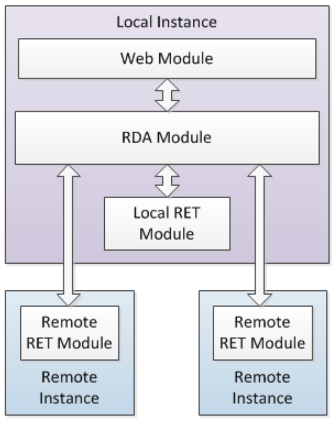

Figure 1: High-level logical system structure.

Request Lifecycle

Every request starts with a RESTful GET call to a designated web service method. The URL used to identify the requested resources has one mandatory path element as well as three mandatory and one optional parameter values – clients consuming this web service are required to specify a name of the request file, start and end times as well as at least one data item name contained in the requested file. The web module submits the request to the RDA component, which queries one or more databases for a list of file paths that fall between the specified start and end time values and match exactly the request file name. At this stage the RDA module attempts to locate each file / resource pairs in either the local or one of the remote caches. Cached file elements are immediately queued on the owning instance for final processing (i.e. culling). The remainder of the file list is sent to a scheduler on a local instance, which attempts to distribute the combined read and process tasks between the clustered RET modules by taking into consideration both performance as well as coherence aspects of the request. The factors, which influence scheduler's decision include:

- The queue length on each cluster instance – the shorter the wait time, the greater the chance that the scheduler will pick that instance.
- Instance spatial location – some cluster member may be physically closer to the stored data than others (i.e. application could be deployed directly on one of the archive server host), which allows them to read stored data at a much higher rate than their networked counterparts.
- Request length – shorter requests, which estimated processing time of under 5 seconds benefit from being scheduled on only one instance.

Regardless of which instance or instances were involved in reading and/or processing the scheduler's request, all intermediate results always make it back to the local instance for final assembly. During this phase of the request lifecycle the results from each processing task are checked for errors. Additionally, if they were processed out of order, they have to be rearranged according to the time index given to them by the scheduler. At this stage the RDA module returns the time sorted, processed results back to the Web component, which can package the processed data in a JAXB compliant wrapper and ship it back to the client.

Spotlight on Culling

Every request that passes through the data server is subjected to the culling algorithm. This algorithm was designed to cut down high volume, time domain datasets to more manageable, lower-density sets. The goal is to return a dataset to the user that is virtually indistinguishable on a scatter plot to the full, un-culled dataset. We have found for typical monitors and resolutions that this occurs when a dataset of about 20k points is returned. The advantage here is that the culled dataset can be transported to the client much more quickly than the full dataset. The disadvantage is that any change in the plot axis limits (for example, a zoom) forces a new data retrieval. Our experience has been that these zoom slowdowns are minimal (< 1 sec) as long as the original data is cached on the data server. The culling algorithm incorporated into the data server uses a parallel processing pipeline, allowing the server to process up to a billion points per second. An enhancement to this algorithm will incorporate user-selected data filters that can be used to further refine the returned datasets.

Hardware

The central EJB container runs in a 64bit environment (Red Hat EL6) on a dedicated rack mounted system. Two 6 core Xeon CPUs, each running at 3.47Ghz, and a total of 144GB of RAM is available to one or more virtual machines - this number depends on garbage collection issues we might encounter once the system is put into production. The SSD caching subsystem is made up of four Intel 250GB solid state drives configured in RAID0 on a 6Gb/s LSI controller. The SSD array is expandable to 3.5TB, and like RAM can be split into multiple partitions. Four 1Gb/s bonded adapters handle the network access to data stores, though only one was used for the results that follow. Satellite servers run in smaller virtual machines on equally powerful CPUs. They have their own in-memory cache, but they lack the disk-store caching, which is available on the core machine. Figure 2 shows the data server's communication diagram.

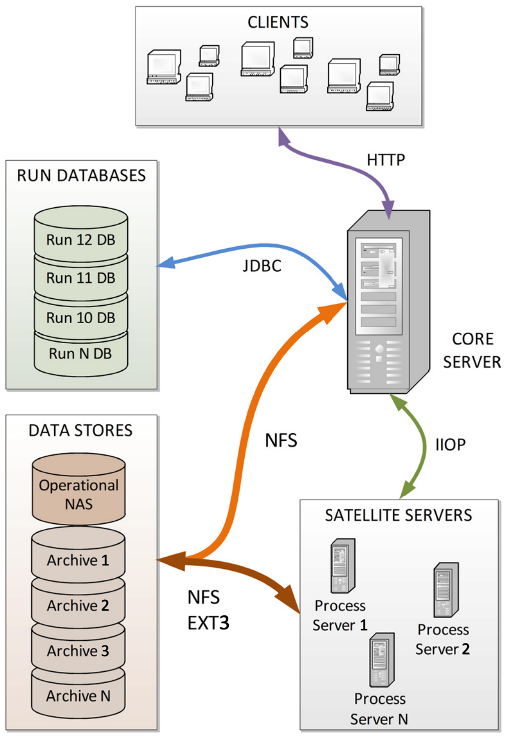

Figure 2: Data server communication diagram.

RESULTS

Data server's parallel processing core is only as fast as the backend, which supplies it the raw data. This is a non-issue for the cached values, however reading from a network store over a 1Gb/s connection can cause a significant bottleneck in the processing pipeline. Before any further development could be done, we had to prove to ourselves that the parallel design could indeed achieve the desired throughput rates. We also wanted to see how the rates scale for both compressed and uncompressed datasets. Figure 3 shows the effective data rates for both types of datasets with varying simultaneous thread count.

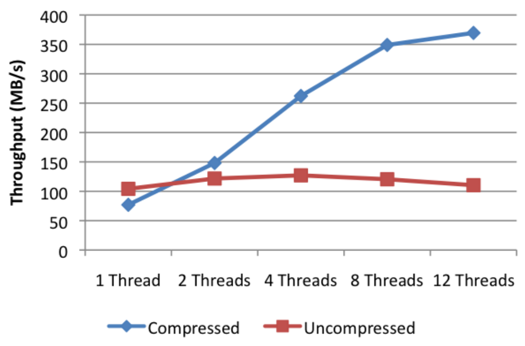

Figure 3: Effective Throughput Read Rates.

The uncompressed data reads over a network are essentially constant around 120MB/s and are unaffected by the additional threads. The same cannot be said for the compressed read scenario. The effective throughput on a 12-core Xeon server increases from 75MB/s to 365MB/s in a fairly linear fashion when moving from 1 to 12 threads. Results for the concurrent compressed scenario were encouraging enough to proceed with the project.

Performance Tests

Our most basic goal for this project was to improve the performance of displaying very large datasets. Our existing software for displaying logged data reads the data directly from disk files and displays the results. Data culling, as described above, is used when necessary to improve display performance. For comparison purposes, this software was modified to allow the option to retrieve data from the data server.

A variety of logged data was tested including very high frequency (10 kHz) and low frequency (1 Hz) data. To eliminate disk-caching issues, a 4 GB dataset was used to flush the cache in between each reading. In addition to having the data server read data from disk files, we also explored the option of reading data from a SSD data cache and from a RAM data cache. The results of these tests are shown in Table 1, which compares the speed at which the displayed data is transported through the various configurations.

Table 1: Data Throughput in Test Configurations

	Throughput	**Speedup**
Client to Remote Disk-store	5.4 MB/s	-
Client through data server to Remote Disk-store	146 MB/s	27x
Client through data server to SSD cache	245 MB/s	45x
Client through data server to RAM	968 MB/s	180x

The large speed advantage of using the data server (in the worst case, a factor of 27x improvement) is the result of the following:

- Network - the data server has a 1 Gb/s connection to the data file vs. a 100 Mb/s connection for the client.
- Parallel Execution - the data server has up to 12 threads that can individually read and cull data files. The client is single-threaded.
- Hardware - the data server has very fast CPUs with lots of RAM in a 64-bit environment. Clients are typically run on much more modest hardware.

The CPU speed turned out to be more important than we originally realized. This is because the data files are gzip-compressed files and need to be uncompressed before the proper data can be extracted. Fast CPUs significantly increase the speed of data decompression.

The results, while reproducible for a specific set of data, were quite variable across different data requests. In fact, the standard deviation across all of the speed tests within each category was about 50%. We have identified several factors that contribute to this variability:

- Data Density - Data for many items are stored together in files, primarily based on how users have setup logging requests. Extracting one or many from the same set of files can significantly affect read performance.

- Timestamp Density - For slow data, we store one timestamp per data point. For most fast data we store data as arrays with a single timestamp and an indication of the time between points.
- Data Compression - Some data compresses much better than others. We've seen data compression rates range from 3 to 30 with 5 being a typical value. More highly compressed data can be read faster from disk because it is smaller.

SUMMARY

At this point it appears that our initial goal of decreasing the time it takes to display large datasets by a factor of 10 will easily be met by introducing a data server between client and data. Much of the speed gain has come from a fast network connection between the data server and the data, fast CPUs on the data server that can be run in parallel to process the data, and a culling algorithm that makes it possible to quickly transport data to the client.

There are several items that we have yet had time to explore. First, we would like to increase the network speed between the data server and our data files. Currently, they are connected via a 1 Gb/s network. We are exploring both 4 Gb/s and 10 Gb/s connections. Second, we would like to make a variety of data filters available to our users so that they can more quickly identify their data of interest. Filter types include "only data when the RHIC collider is on its energy ramp", "only data when there is beam in RHIC", and "only data when a measurement is above/below a threshold value". These filter selections will be passed down to the data server and reduce the amount of data that the server needs to process and the associated data that the user will view. Finally, we have plans on making better use of the solid-state drive connected to our data server. Past experiments have shown that users spend about 80% of their time looking at logged data collected within the last week and 90% looking at data collected within the last month. Our plan is to store recent logged data on our 1 TB SSD so that these frequent requests can be delivered more quickly.

REFERENCES

[1] K. Furukawa, M. Satoh, I. Mejuev, K. Nakao, "A Java-Based EPICS Archive Viewer With Soap Interface For Data Retrieval”, http://accelconf.web.cern.ch/AccelConf/ica03/PAPERS/MP707.PDF, ICALEPCS (2003).

[2] Roderick, C., UK Oracle Users Group Conference, http://lhc-logging.web.cern.ch/lhc-logging/docs/Presentations/LHC_Logging_Service_UKOUG_2006.ppt (2006)

[3] Oracle Corporation, “Glassfish Metro Users Guide”, http://metro.java.net/guide/

[4] Oracle Corporation, “JSR-000316 Java Platform, Enterprise Edition 6 Specification 6.0 Public Review Draft”, http://download.oracle.com/otndocs/jcp/javaee-6.0-pr-oth-JSpec/