SOFTWARE DESIGN CONCEPTS FOR ARCHIVING AND RETRIEVING CONTROL SYSTEM DATA[♣]

C.A.Larrieu, M.H.Bickley, TJNAF, Newport News, VA 23606, USA

Abstract

To develop and operate the control system effectively at the Thomas Jefferson National Accelerator Facility, users require the ability to diagnose its behavior not only in real-time, but also in retrospect. The new Jefferson Lab data logging system permits the acquisition and storage of enough data to provide suitable context for such analyses. In addition, it provides an extraction and presentation facility capable of efficiently fulfilling requests for both raw and processed data. This paper discusses some of the design goals and implementation decisions involved in the development of "CZAR", the **C**hannel Access **Z**ippy **Ar**chiver. Among these are: extensibility and maintainability via object-oriented and compartmental design, reliance upon a relational database system for storing configuration and data summary information, integrated support for common statistical and filtering transformations in the Application Programming Interface (API) which developers use to access logged data, and the use of CORBA to facilitate deploying the system in a heterogeneous environment.

1 GOALS

The design of the Jefferson Lab archiving system follows from two primary goals: foremost, that it fulfill the operational needs of Jefferson Lab; secondly, that it benefit other members of the EPICS collaboration either in part or in toto.

1.1 Data Acquisition

Ideally, a control system archive should contain all the information necessary to construe an accurate representation of the state of system parameters at any time in the past. For a digital control system, one way to achieve this goal is to acquire and store every value of every control point at every discrete instant in time. Unfortunately, such a strategy does not scale well: as the number of control parameters and the rate at which they change increases, performance requirements unavoidably begin to strain network and storage capabilities.

A more practical approach is to gather and store only those data which contribute significantly to the state of the control system at any particular time. But this technique also poses a problem, since the importance of certain parameters may vary depending upon context within the control system.

The CZAR provides an infrastructure which gives system experts as much fine control as possible over how and when the engine should gather and store specific machine parameters. It does so by delegating data acquisition tasks to DAQ modules. If one of the standard DAQ modules does not provide sufficient control, a C++ developer can create a new module with relatively little work.

1.2 Data Retrieval

In order to benefit from having stored control system data, users require the ability to examine it in some useful fashion. Typical uses for such data include correlating values in time, searching for trends, generating status audits, etc. These activities may occur interactively or batched. For example, an expert may browse through archived data while debugging a system, whereas another user may wish to generate periodic reports summarizing pertinent characteristics of the control system over a certain time range.

By providing a data retrieval API which incorporates standard data transformations, the Jefferson Lab archiver simplifies the task for developers who wish to create tools for interacting with the archived data. Furthermore, by providing this functionality in a library built on top of a narrowly defined archiver access API, it becomes available to other laboratories who wish to implement the portability layer on top of their own archiver instead of using the CZAR.

1.3 Usability

An important, but often neglected aspect of software design is consideration of how users interact with it. One significant design goal of this archiving system is to provide simple user interfaces for managing all aspects of its operation. All components of the CZAR system incorporate built-in support for user feedback via a C++ interface which provides virtual methods for informing the user of the current task stack and activity

[♣] This work was supported by the U.S. DOE contract No. DE-AC05-84-ER40150

level, as well as the overall progress towards its completion.

Defining CORBA interfaces to the major components of the system facilitates the task of adding platform-independent distributed interfaces as the need arises.

2 IMPLEMENTATION

The entire archiving system actually consists of several distributed components:

1. A relational database which stores configuration information and archive "metadata".
2. A data acquisition engine which gathers real-time control system data and caches it in an intermediate staging area on the filesystem.
3. A daemon process which controls the engine, and periodically converts staged data into a compressed long-term storage format.
4. A server process which fulfills requests for history data.
5. A graphical interface for inspecting and editing the archiver configuration.
6. A Client library for accessing and manipulating stored data, and associated end-user tools.

2.1 Relational Database

Common practice, especially in UNIX environments, has traditionally favored storing configuration information in plain-text files, the primary motivation being to ensure easy access to the data via text editors and file-system utilities. But this technique requires either exporting the file-system or developing a special-purpose server to share information with distributed components. By using a database, we gain network-transparency in a fairly standard fashion. The database also helps to ensure data consistency by synchronizing access from distributed components.

With the exception of the actual data points acquired from the control system, CZAR stores all of its information in a relational database (currently MySQL). This information includes the network and file-system location of archive processes and data, start-up options, data acquisition specifications, and administrative information.

The database also stores summary information for logged data, including data acquisition begin and end times, number of points, data format, file name, and file status. It stores Channel Access control information (data type, engineering units, alarm limits, etc.) for every logged signal, and maintains a connection history for each. It provides a fast directory into the file system for finding the actual data for a specific signal and time range. We chose not to store the actual logged data in the database because doing so might exceed database table limits and would complicate the task of transparently shuffling data from on-line to off-line storage (i.e. from local file-system to tape silo). Furthermore, accessing binary data from the file system is more efficient than accessing it via the database.

In general, our strategy has been to use the database for those tasks to which it is well-suited, capitalizing upon its inherent suitability for use in a distributed system.

2.2 Data Acquisition Engine

The data acquisition engine runs as a child of a daemon process, and is controlled by its parent via pipes (to be reimplemented with CORBA). When the engine is activated, it retrieves from the database the list of all signals to log. For each of these, it then retrieves the data acquisition specifications, also from the database, finds the appropriate DAQ module by name, and then dispatches responsibility to the appropriate object which can either be a compiled-in object, or a dynamically-loaded shared object module.

The two standard, compiled-in, data acquisition modules implement the "monitor" and "trigger" techniques. The former captures every change of its associated signal which occurs outside a specified time interval. The latter allows changes in certain "trigger" signals to initiate a countdown timer which will cause the buffers for dependent signals to flush. By specifying a suitable buffer size and delay time, a user can capture the state of various control points for a period before and after some event. This can be quite useful in diagnosing machine behavior.

To implement a new module, a programmer must create a derived class of a generic DAQ module parent class, implementing several virtual methods which allow interaction with the user at configuration-time and control by the engine at run-time. While the process is fairly simple, it does require that the programmer understand how to capitalize upon the facilities which the engine provides, such as interval timers, asynchronous event handling, channel access ring buffers, and the output subsytem.

All output is written to a pair of files. One of these is an index into the other. Whenever a data buffer for a signal is committed to disk it is first written to the data file and then indexed in the index file. This approach lends the system a modicum of fault-tolerance. If the engine should terminate for some reason (e.g. machine reboot, run-time exception), the possibility of the output data becoming corrupt depends upon the possibility that an index entry becomes corrupted, which is quite unlikely. When the data file reaches a pre-defined size, the engine creates a new file pair and redirects all subsequent output. The list of files created by the archiver engine is enumerated in a log file in the output directory. This file also stores information about engine run-time eents, such as when and how the engine is started and stopped.

2.3 CZAR Daemon

The process which activates and deactivates the data acquisition engine also periodically converts the engine's output data into a more efficient long-term storage format, updating the data summary information in the database. Because the original output format is designed for fast output and fault-tolerance, it contains a good bit of redundant information which the conversion process discards. The converter also performs some simple compression during its operation, which generally yields between 50% and 75% reduction is storage requirements.

2.4 History Server & Client API

The server process also runs as a child of the CZAR daemon. It accepts requests for specific ranges of data, loads the corresponding data from the filesystem, then returns it to the client. It also accepts and acts upon requests to convert portions of data from stage format.

Centralizing access to the data reduces the amount of specialized knowledge client applications need in order to retrieve archive data. Standard data reduction and transformation operations implemented at the server can also yield a net performance gain, especially if the resulting data set is smaller than the input data set. For example, if a client wishes to average a year's worth of data, the server can perform the averaging and return the summary, instead of sending all of the data over the network.

The client API is comprised of several layers. At the lowest level it consists of the collection of classes from which the engine is built. In order to coalesce this somewhat disorganized set of objects and methods into a coherent interface, CZAR implements the "CADataStore" interface [1], which is intended to encapsulate the essential capabilities of any repository of channel access data. As such, it represents a simple portability layer between client applications and the archive data storage system.

2.5 Configuration Editor

This is a Java application which uses JDBC to communicate with the database, and which presents the user with tools for manipulating the archiver configuration. Specifically, it allows for changing the DAQ specs for groups of signals, adding new signals, deactivating old signals, changing administrative groupings, creating new groups, etc.

The standard transform and data manipulation library is built upon the CADataStore interface, and provides access to common statistical and filtering operations. For example, it implements binning and interpolating operations for reducing data at the server. By providing these routines as part of the access API, we hope to simplify the task of developing intelligent client applications.

3 REMAINING WORK

4.1 CORBA

Currently, the distributed components communicate via a simple TCP/IP protocol. They will be modified to rely upon CORBA, which will in turn yield several significant advantages. First, they will no longer be architecture dependent, since CORBA handles data type conversions between native formats. This in turn will facilitate distributing the components among heterogeneous machines. Specifically, it will facilitate the development of a Java-based data analysis tool.

4.2 Platform Porting

So far, all development and testing has transpired in an HP-UX 11 environment. Once the system has reached release quality it will be ported to Linux and possibly Solaris.

4.3 Specialized Compression

Currently, the permanent storage format uses the zlib compression library to code chunks of data. While this results in typically around 50% compression, preliminary research indicates that a specialized algorithm using adaptive and predictive arithmetic coding [2] will reach about 95% compression. This is significant because we are currently constrained by limited storage capacity.

ACKNOWLEDGEMENTS

Many of the design requirements and implementation issues discussed in this paper arose from discussions with various members of the EPICS collaboration, especially Bob Dalesio and Kay Kasemir at Los Alamos.

REFERENCES

[1] C.A. Larrieu, "EPICS History Server", http://www.jlab.org/~larrieu/work/History/history.html.

[2] C.A. Larrieu, "Compressing Control System Data at Jefferson Lab", http://www.jlab.org/~larrieu/work/Compress/paper.ps