Data Handling and Filter Frameworks for the D0 L3 Trigger System

A. Boehnlein[1], G. Briskin[2], G. Brooijmans[1], D. Claes[3], D. Cutts[2], S. Mattingly[2], M. Souza[4], G. Watts[5].

[1] Fermilab
[2] Brown University
[3] University of Nebraska
[4] LAFEX/CBPF - Rio de Janeiro
[5] University of Washington

Abstract

The D0 Collaboration at the Fermilab Tevatron will use a farm of commodity PCs running the Windows NT OS as the third level trigger system. Data blocks from front-end crates will move independently in a data driven mode to individual nodes of the L3 farm. The input rate into the L3 farm is planned to start at over 1 KHz.

The data-handling framework located in every L3 farm node is built from separate components that will load and lock events into physical memory, examine event integrity and facilitate in the physics filter operation. Multiple events will be staged in memory, and each node will use SMP architecture to allow for multiple filter processes.

The L3 filter is software-based and will use all available information from the D0 detector. The trigger decision will be based on partial event reconstruction, where the target rejection for this trigger level is a factor of 20 with an average decision time of 50 milliseconds. The steering code for this partial reconstruction is written in C++ and uses object oriented design techniques and patterns. It includes flexible on-demand dynamic loading of the reconstruction components based on the trigger menu, as well as centralized statistics and error management systems.

In this paper we report on the design, implementation and status of the data-handling and filter frameworks.

Keywords Trigger System, Filtering, Level 3

1. Introduction

A Level 3 farm node is a combination of two separate hardware components, a multi-processor machine with a Network Interface and a VME crate with MultiPort Memory (MPM) boards. The event fragments from Front End Crates (FECs) arrive into the MPM boards located in the VME crate. This is the first time in the DAQ system that all the data for a single event comes together. The assembled event data is then loaded into the analysis node memory. A L3 farm node will be a high-end SMP machine. The exact configuration will be determined as close to the beginning of the Run 2 as possible in order to take full advantage of the continual hardware improvements and price drops in the commodity market.

The L3 Node Framework software has been designed to manage the data flow in and through the L3 Farm Node. The framework is responsible for configuring the MPMs for the arrival of a new event, loading event data into processor memory, and checking event-data integrity. The event data is then distributed to one of several L3 Filter processes for physics analysis. If the filter process accepts the event, it will be sent to the online system for recording and distribution; otherwise the event data will be discarded.

The L3 Node Framework is designed to run on the Windows NT operating system (OS). Windows NT has been designed from the ground up to be a highly responsive, general-purpose OS.

In the following we will discuss the different components of the L3 Node Framework and L3 Filter Process.

2. Data Handling Framework Software

Figure 1 shows the internal framework structure. The lines represent the logical data flow through the framework software components. Every component is a separate thread of execution. The ladder boxes in the data flow path represent queues that hold pointers to the events in shared memory. The use of event pointers is dictated by the requirement that no copying of event data should take place after it is loaded into the memory of a farm node. The queues make communication and synchronization between framework components simple and straightforward. With this approach software design is natural and modular: it is easy to plug in a new component without modification to the framework. For the most part, all components are represented by threads and exist inside a single Windows NT process. The exception is the L3 Filter; it runs as a separate process in its own address space. By isolating the DAQ code from the physics filter code we can build and release the two modules independently of each other. Furthermore, by isolating the filter process we prevent crashes in the filter code from bringing down the whole farm node. This is especially important, as there may be more than one L3 filter process running on a single farm node. The event data the filter process was working on when it crashed will be flagged and then passed on to the host system.

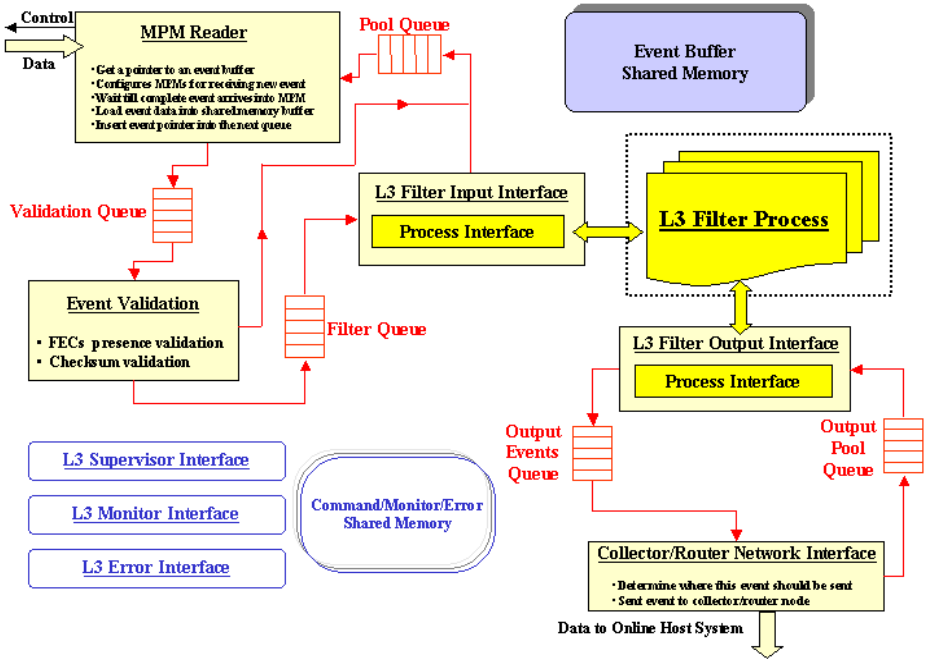

Figure 1. L3 Node Framework Software.

- **MPM Reader Active Object**: The MPM Reader Active Object controls and uploads data from the MPM boards located in the attached VME crate. The upload of event data into

analysis node memory is done via the bridge DMA engine. While DMA is in progress (i.e. copying of event data from MPM buffers into node memory) the analysis node processor(s) are free to perform other L3 framework tasks.

- **Validation Active Object:** The Validation Active Object examines raw event data for any errors. It insures that all the Front End Crates (FECs) that were supposed to send their data for this *event class* are present in the raw event. If raw event data is corrupted in any way it will be dumped or copied and sent to a designated node for debugging purposes. After appropriate action, in case of validation failure, the event pointer is returned to Pool Queue. Otherwise, the event pointer is added to the filter queue.

- **L3 Filter Input Interface Active Object:** The L3 Filter Input Interface Active Object communicates with the L3 Filter Process. The inter-process communication (IPC) mechanism between the node data-handling process and the L3 Filter process uses shared memory and an Event Synchronization Object. The number of L3 Filter Input Interface Objects is equal to number of L3 Filter processes. If the L3 Filter process crashes for any reason, the raw event data will be dumped or copied and sent to a designated node for debugging purposes. The pointer to an event buffer will then be returned to the Pool Queue and the crash reported. The L3 Filter Input Interface Object then restarts the L3 Filter Process.

- **L3 Filter Output Interface Active Object:** The L3 Filter Output Interface Object is notified by L3 Filter process that an accepted event is available for transfer to the online system for storage and distribution. The number of L3 Filter Output Interface Objects is equal to the number of L3 Filter processes. If L3 Filter process crashes for any reason the pointer to the output event buffer will be returned to the Output Pool Queue.

- **Collector/Router Interface Active Object:** The Collector/Router Interface Active Object sends filter-accepted events to the online Collector/Router for distribution to other online system components. The wrapper inserts required information into the header prior to sending the event. In the case of network failure the Collector/Router Interface Object will try to reconnect and send the event for a specified number of tries. If the object fails to reconnect and send successfully, the output data will be stored on the local disk and network error condition will be reported. When the network connection is reestablished the locally stored events will be sent to the online system.

- **L3 Filter Process:** The L3 Filter process provides a main entry point into the physics filter analysis program and is described in greater details in the next section.

3. L3 Filter Framework Software

L3 Filter is a separate process and runs in its own address space. From the Run 1 experience it was learned that the physics filter code changes more often than other parts of the framework. Isolating the filter process to its own address space prevents memory corruption in the data-handling process and/or crashes in the filter code, from bringing down the L3 Farm Node. In addition it makes integration of new releases of the filter code into the L3 Node Framework easy and straightforward.

The L3 process framework is responsible for inter-process communication with the data-handling process and in providing a correct calling sequence to the filter framework public

interfaces. It is also responsible for keeping track of all messages that need to be passed to the filter framework for correct operation. The L3 filter process' access to raw event data will be through a read only memory. In the following we describe in greater detail D0 filter framework infrastructure.

3.1. ScriptRunner

ScriptRunner[1] is the set of classes that handles the general Level 3 infrastructure. One function is to supply the "hooks" into the Level 3 Framework (described above). This allows for the control of the filtering process, from initialization of run conditions, start of run, through end of run and the accumulation of monitoring and error messages. Under proper *hooks*, ScriptRunner reads in the trigger list and based on that trigger list creates a map of the available tools and filters and then an execution tree. ScriptRunner is then responsible for steering the event by event processing, and for returning the trigger decision to the framework. ScriptRunner will check all events received for the passed Level 2 bits and/or marked as unbiased. Based on the fired trigger bits, ScriptRunner will traverse the corresponding execution tree branch, execute the appropriate filter scripts, which execute Level 3 filters, which in turn execute the needed tools. Tools and filters must keep track of their results, in case they are called more than once per event. All filter scripts of all fired L2 bits must be run to come to a decision. At the end, the event is either passed, failed, unbiased or in error.

For passed and unbiased events, ScriptRunner requests a copy of the relevant tool and filter information to put into the L3 output chunk which is added to the event data. Then ScriptRunner notifies the tools and filters to reset, and returns the decision to the framework, which then takes appropriate action based on the decision.

3.2. Tool and Filters

The functionality of providing a list of candidate physics objects is separated from the actual trigger selection. Tools are classes that implement the algorithms for identifying candidates, while filters check whether any of the candidates satisfy the trigger conditions.

3.3. Tools and Filter Registration

Tools register their address with ScriptRunner, using a Factory pattern (ToolFactory), through an appropriate C pre-processor (Cpp) macro in a similar manner as the D0 offline batch framework registration scheme[2]. This procedure isolates the Tool developers from the bookkeeping, while allowing a straightforward implementation of Tools as dynamic linked objects. This Cpp macro takes the Tool type as parameter and creates pointers to prototype tool objects, from which copies, keyed by their names, will be made upon request. For instance, *ele_loose*, *ele_tight*, *ele_escape*, are different instances of the Electron Tool type. Through its Macro, the first Tool of type Electron that is instantiated calls the prototype maker and makes an instance of it. All others make copies (clones) of this Tool type, with different names. They are unique as their (pseudo) constructors reads distinct parameter sets.

Every time a Tool instance is made, the ToolFactory performs several functions:

- A new entry to this Tool type is added to the pointer map for the first registration of a given Tool type.
- For each Tool instance, if the first time
  - a new entry in the Tool instance map is made -
  - and a new entry to the reference count map for this Tool name is made.

- Otherwise,
  - The reference count is incremented for this instance.
  - This Tool instance address is returned from the instance map.
- DLL information for this Tool instance is registered - see below

Currently, Filter registration is hard-coded into the filter map, but a flexible registration scheme will be implemented in the future.

3.4. Dynamic Linked (DLL) Tools

In principle, the Tools should only be loaded when needed by the current trigger programming. Conversely, Tools should be freed when they are not needed anymore. But that could cause a bookkeeping overhead for the L3 framework, if each Tool has a separate own DLL. The current implementation allows this *strong* DLL choice as well as an intermediary situation, grouping Tools in a few DLLs by functionality or just have one DLL for normal data collection and others for test purposes and so one. The method chosen was that the Framework will pass ScriptRunner the DLL file name where a given Tool sits, when such Tool is requested (dynamic linked). In this way, one can make any grouping choice and even modify it. The fact that Tools can call others Tools is transparent.

The main ingredients are:

- All Tools belong to an abstract base class to which an exported symbol is created. In this way, all individual Tools are searched by the C++ compiler/linker itself through its virtual tables, instead of the usual (cumbersome) DLL link tables. The Tool registration method described above implements this scheme. From the NT DLL machinery, only two functions are needed:
  - handle = LoadLibrary(dllfilename);
  - FreeLibrary(handle);
- A welcome consequence is the use of the Unix's calls counterpart dlopen(dllfilename) and dlclose(handle), making the NT - Unix switch very straightforward.
- We use our own reference counting for each DLL that is downloaded, instead of the automatic system wide one. That is, we ``LoadLibrary" and ``FreeLibrary" only once, the other calls are ``reference counted". This allows the possibility to have several ScriptRunner executables running in parallel in EACH node. The system wide DLL automatic reference counting will deal only with references to a same DLL in more then one ScriptRunner copy. This insures, as usual, that only one copy of a given DLL will be loaded in each node.

References:

[1] "The D0 Level 3 Software Trigger", Amber Boehnlein, Dan Claes, Moacyr Souza, Gordon Watts**,** *CHEP '98: Proceedings of the Conference on Computing in High Energy Physics*, Chicago, (1998)

[2] Jim Kowalkowski, ``D0 Framework,", http://home.fnal.gov/~jbk/docs/index.html}, 1998.