CMS Online Event Filter Software

Andrés Kruse [a]

[a] *CERN, ECP division, 1211 Geneva 23, Switzerland*

The CMS trigger system consists of a two stage system: A first level trigger system and a large farm of processors performing the online event filtering. The presented paper gives an overview of the current status of development of the filter farm software. We present an architectural model for efficient filter farm software and show a multi-threaded event filter prototype.

1 Introduction

The CMS experiment [1] is one of the two LHC general purpose experiments. The detector is scheduled to start data taking in 2005. The design of the CMS event filter farm is an ongoing process, many of the details will only be settled during the next years. Both hardware and software implementation details will change throughout the lifetime of the experiment. Hardware design concentrates on finding the building blocks of a scalable system. Software design must construct an architecture which supports scalable solutions and in which the key abstractions and mechanisms are unlikely to change. Object oriented technology supports this approach.

2 CMS Trigger System

2.1 First Level Trigger

The CMS online trigger system consists of two levels. The first level trigger operates at LHC frequency of 40 MHz and is expected to accept events at a rate of up to 100 kHz. Events are read out and buffered on about 500 readout units. The event size is in the order of 1 MByte.

Preprint submitted to Elsevier Preprint *17 July 1997*

2.2 Online Event Filter Farm

All readout units are connected through a large switching fabric to a farm of event filter nodes, high performance workstations on which the filtering task is performed (see figure 1). The required processing power of the farm is $5 * 10^6$ MIPS which is expected to require about 10^3 processors in the year 2005.

Monte Carlo studies show that a large fraction of the events can be rejected by algorithms which only access the data of a subset of the readout units. A positive decision however usually requires access to all data. If only the necessary data are transported through the switch the required bandwidth can be reduced significantly from about 1000 Gbit/sec to 200 Gbit/sec and thus reduce costs of the switch fabric.

It is impossible to predict which portion of the data will be accessed by a filter algorithm for a particular event because this depends on the event data itself. Consequently the event filter farm cannot operate in "push mode" where only the required subset of data are sent to a node. Instead it works in "pull mode": A filter node starts with L1 trigger information and can request more data as it evaluates the event. As soon as a negative decision has been taken the event is flushed from all buffers. For a positive decision the remaining event data are requested and the full event stored in the offline database. The maximum output rate of the event filter farm is 100 Hz, one per-mille of the incoming rate.

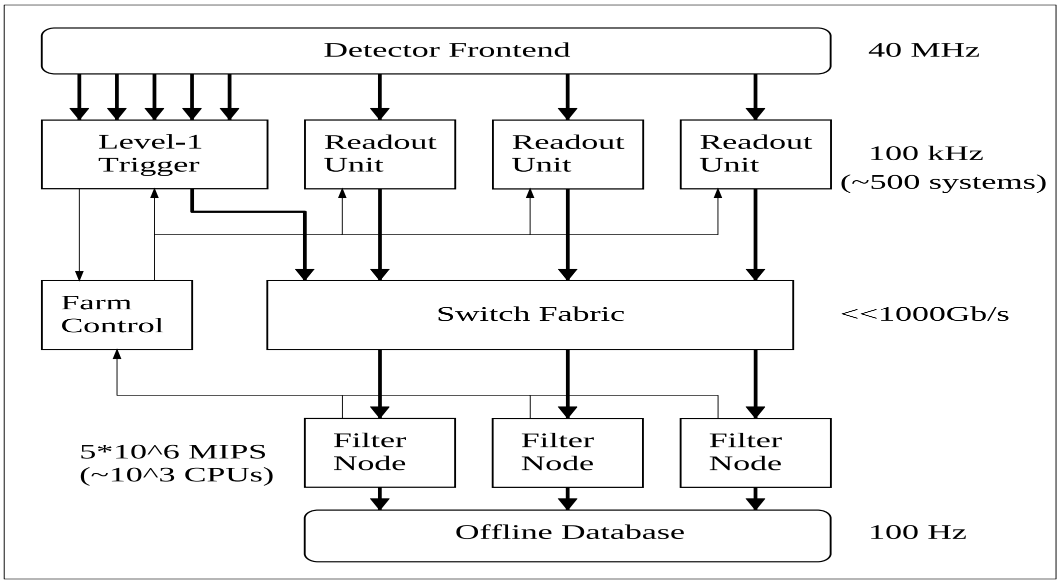

Fig. 1. CMS Online Event Filter Farm

3 CMS Event Filter Farm Software

3.1 Object Oriented Software Engineering

CMS offline software is object oriented and currently engineered using the Booch method. Online software is developed with the same method. The software life cycle follows an evolutionary approach. Through prototypes the software architecture is verified. In every development cycle emphasis is put on key classes and their interaction.

3.2 Software Requirements

Key features of relevance for the design of the main filter functions are:

- The filter algorithm must not assume to find all event data when it starts processing. Instead it has to request the necessary data which are distributed over many readout units.
- Each filter farm node performs its main function independently of the other nodes.

There are additional requirements. CMS online software must:

- make efficient use of the bandwidth and computing resources available.
- run offline filter algorithm code without modifications.
- be of high quality: Software errors will compromise the physics performance of the CMS detector.
- be able to handle failure of subsystems.
- allow for a reconfiguration into partitions to cope with the need for independent detector studies.
- support online calibration and monitoring of the detector during datataking.

3.3 Reconstruction on Demand

The CMS offline object model is based on “reconstruction on demand” (see [2]). An object, like a track or jet is not instantiated a priori during “reconstruction processing” but only when the object is being requested for the first time. This is done transparently without knowledge of the requester. Object oriented technology permits an implementation with only one additional indirection and so this mechanism will limit the amount of CPU time spent on an event to the minimum necessary.

3.4 Data Retrieval on Demand

The mechanism of "reconstruction on demand" can be reused for the filter farm software: There is no pre-emptive request or delivery of data except for the L1 trigger information. When all L1 information for a new event has arrived the event becomes accessible and is stored in a FIFO. The filter picks it up from there and starts processing.

When the filter algorithm accesses raw data of a readout unit for the first time, the data are requested and the filter blocks, waiting for the data to arrive. It is up to the implementation to decide whether immediately the raw data of the entire subdetector is requested or whether a request can also be performed on readout unit level. When the data have arrived the application continues. Through object oriented technology this mechanism can be fully transparent, to the filter algorithm the data always appear to be present.

3.5 Multiple Workers

The services provided by todays operating systems like multi-processing, multi-threading and signal-handling allow to create many independent worker threads on an event filter node. Each worker handles one event at a time. It performs the filter algorithm as a single thread of execution. While waiting for requested data to arrive, a different worker can continue processing its filter algorithm. All workers share a single resource, the interface of the node to the switch fabric. It is useful to structure the software into a synchronous layer in which the workers execute the filter algorithm and an asynchronous layer in which lower level tasks perform the I/O.

This is also known as the "half-sync/half-async pattern" [4]. Every modern operating system implements such a pattern, thus simplifying the programming of higher level tasks sharing a single resource.

3.6 Raw Data Server

The asynchronous I/O of an event filter node is handled by a "raw data server". It communicates with both the workers and the switch fabric. It stores new events in a queue which can be read by workers. It accepts requests for more data and forwards the request to the switch fabric. It receives the data and wakes up the worker that is waiting for them (see figure 2). The raw data server can be implemented as operating system driver, separate process or as passive object within a multi-threaded application.

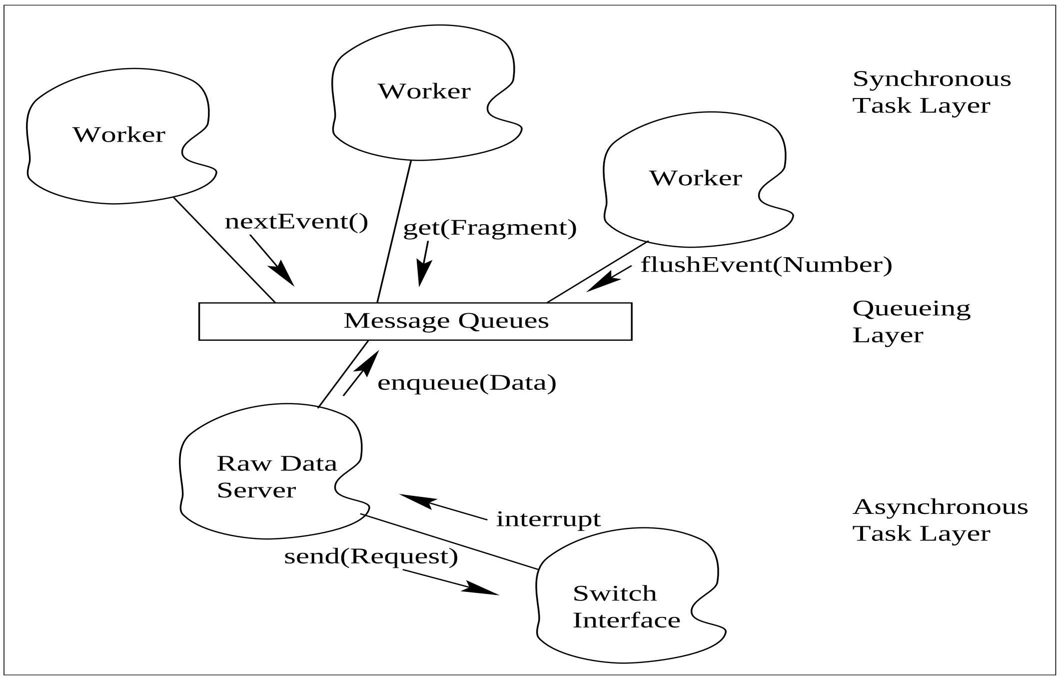

Fig. 2. Structure of the Event Filter Node Software

3.7 A Multi-Threaded Filter Prototype

A prototype of the filter farm software has been implemented using a multi-threaded approach (see figure 3). The prototype is used to study techniques and patterns necessary in a multi-threaded environment.

Here every *Worker* is an active object [5] which executes its own light-weight thread. One of the problems to be solved is the implementation of the "queueing layer" shown in figure 2. In this layer the communication between the synchronous *Worker* objects and the asynchronous *RawDataServer* is handled.

The *NewEventQueue* is implemented as a list object. A mutual exclusion mechanism is used to protect the list from being modified by several threads at the same time. Through a condition variable a *Worker* can block if the list is empty.

To request more raw data for an event, a *Worker* invokes the method *getData()* on the *RawDataServer* interface. The *Worker* thread does not block inside *getData()*. Instead a handle on the pointer to the "to be delivered" raw data, a "Future" object, is returned and the *Worker* can continue. Only when it wants to get the pointer from the *Future* it will block. When the data have arrived and the pointer to them has been passed to the *Future*, the *Worker* thread is unblocked and continues. A *Future* implements a "single write, multiple read" pattern. It permits asynchronous method invocation between active objects (see [5]).

Through the future mechanism this prototype is able to handle multiple outstanding requests of data per event: The filter algorithm could itself be multi-threaded.

C++ code for a polymorphic future was developed for this prototype and is now part of the ACE class library [3].

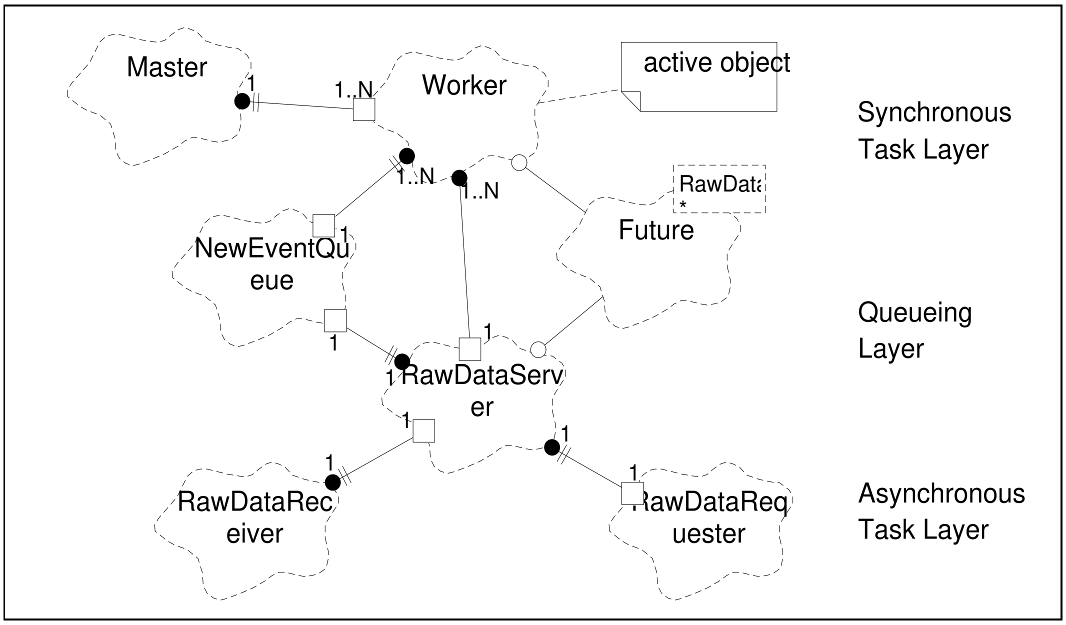

Fig. 3. Multi-Threaded Filter Node Software Prototype: Class Diagram

3.8 Model Characteristics

The software architecture presented in this paper has the following characteristics:

- Event building is part of the event filtering task.
- Resources are used when they are needed. It is the filter algorithm and the hardware resources which determine the throughput of the system.
- The pull mode in which the software operates might result in large latencies. This is not considered a problem as at startup time of CMS very large buffer sizes will be affordable.
- Should switch technology and processor performance increase dramatically the filter farm can operate in "push mode", the software architecture does not have to be modified. Even in this case the presented mechanisms have advantages because bandwidth and processing power are only used when needed, e.g. raw data are not unpacked pre-emptively.

4 Event Filter Software Prototypes

In 1996 software prototypes were used to study techniques, to get acquainted with object oriented software development and to develop the patterns described above. Development in 1997 concentrates on a prototype for a CMS testbeam. Here a full chain from online data acquisition to offline analysis will be set up. The online software will be used as a test bed for techniques described in this paper. It will be developed using the LHC++ library, the OOA&D tool Rose/C++, the ACE class library [3] and the Objectivity data base. Later laboratory prototypes of a small scale switch fabric will be used to test the scalability of the proposed solutions and to evaluate different switch technologies.

5 Summary

We described possible mechanisms and methods for efficient use of the computing resources and exploitation of the bandwidth available in the CMS event filter farm. It was shown that event filter and event builder can be integrated into one system and that the CMS offline object model fits into the online event filter farm framework. A prototype based on multiple lightweight threads was presented. It was shown how object oriented technology helps to find the key abstractions in the filter farm software. An overview over the current and future developments of CMS online event filter farm software was given.

References

[1] "CMS - The Compact Muon Solenoid, Technical Proposal", CERN/LHCC 94-38, LHCC/P1, CERN 1994

[2] "The Compact Muon Solenoid, Computing Technical Proposal", CERN/LHCC 96-45, CERN 1996

[3] Douglas C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications", *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, April 1994

[4] Douglas C. Schmidt and Charles D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O", *Pattern Languages of Program Design, Addison-Wesley, Reading, MA*, 1996

[5] R. Greg Lavender and Douglas C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming", *Pattern Languages of Program Design, Addison-Wesley, Reading, MA*, 1996