A Software Approach for Readout and Data Acquisition in CMS

G. Antchev[1], E. Cano[1], S. Chatellier[1], S. Cittolin[1], S. Erhan[2], D. Gigi[1], J. Gutleber[1], C. Jacobs[1], F. Meijers[1], R. Nicolau[1], L. Orsini[1], L. Pollet[1], A. Racz[1], D. Samyn[1], N. Sinanis[3], P. Sphicas[1]

[1]CERN, Geneva, Switzerland; [2]University of California, Los Angeles, USA; [3]ETH Zurich, Switzerland

Abstract

Traditional systems dominated by performance constraints tend to neglect other qualities such a maintainability and configurability. Object-Orientation allows one to encapsulate the technology differences in communication sub-systems and to provide a uniform view of data transport layer to the systems engineer. We applied this paradigm to the design and implementation of intelligent data servers in the Compact Muon Solenoid (CMS) data acquisition system at CERN to easily exploiting the physical communication resources of the available equipment. CMS is a high-energy physics experiment under study that incorporates a highly distributed data acquisition system. This paper outlines the architecture of one part, the so called Readout Unit, and shows how we can exploit the object advantage for systems with specific data rate requirements. A C++ streams communication layer with zero copying functionality has been established for UDP, TCP, DLPI and specific Myrinet and VME bus communication on the VxWorks real-time operating system. This software provides performance close to the hardware channel and hides communication details from the application programmers.

I. Introduction

High Energy Physics experiments produce large amounts of data that have to be read, processed, and stored persistently for later analysis [1]. A split of responsibilities into readout and processing can help to concentrate on the critical issues of the two parts in on-line processing. As the requirements on the detector readout are demanding in the area of data transfer and buffering capacities, several technologies have to be evaluated before concrete design and implementation phases for the experiment can take place. Such a prototype driven process is supported by the prevalence of combinable hardware building blocks: processor boards, network interface cards, memory devices and standard bus systems. There exist a number of different physical data transfer and processing components in the prototypes for the Compact Muon Solenoid (CMS) experiment readout system under study. Although efficient, low level access of these components from within specialised application programs would increase coupling at the expense of flexibility. We address this problem by applying a Component Based Software Engineering approach [2],[3],[4],[5]. For each hardware component, a corresponding software image exists. Differences in accessing the hardware are smoothed by software glue. The object-oriented approach allows abstracting out the complexity of the functions and data involved in the operations and relieves us from having complicated parameterisation schemes. Thus, a uniform view of the data transport layer can be presented to the application programmer [2],[6].

II. Motivation

From prior experiments in high-energy physics we have learned that such installations are characterised by a long lifetime. During this time the system will be subject to incremental upgrades, that is changes or replacement of hardware components, in order to fulfil evolving requirements. Experiments that are currently under design for next generation colliders like the Large Hadron Collider, require extensive prototype and design phases in which the set-up with the best cost/performance ratio has to be identified [7]. In this context, when composing a concrete system we are faced with strong requirements for flexibility and re-use. Because of their nature, read-out systems are characterised by data movement. Input and output operations from and to devices are the main function of such systems. With regard to the diverging interfaces of the hardware components involved, special software is needed to logically connect them [8],[9]. The goal is to provide a software framework that isolates the developer from software layers that interact heavily with the physical environment, and maintain and meets the need of the program specification.

III. The Readout Unit in CMS

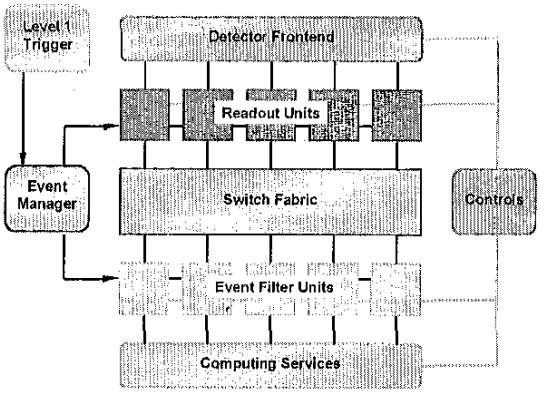


Figure1: The CMS data acquisition system structure

The average data volume generated by the CMS detector is expected to be of the order of 1 MByte with a maximum event rate of 100 kHz. However, the final output of the

experiment should not exceed 100 events/sec [7]. The approach to data acquisition systems that is being investigated in CMS to handle this stream of information is to make a physical distinction between the acquisition and the processing parts, see Figure 1. The system can be described with a generic client/server model, which is made up of two distinguishable entities, the Filter Unit (FU) and the Readout Unit (RU). The FU is the client and requests services from the RU that is the server. In particular, the RU receives event data fragments from detector dependent units on dedicated links; it buffers them locally and provides them on demand to the FU over the available interconnection network. The flow of event data will be controlled by an event manager system. Internally the RU is split into its functional responsibilities. A readout unit input (RUI) for handling data coming from the detector, a readout unit output (RUO) for serving requests from processing units and a readout unit supervisor (RUS) for control, monitoring and configuration. There exists also a special readout unit memory (RUM) device [10] for buffering the data from the detectors (see Figure 2). The RUI and the RUO represent two tasks that operate independently of each other. The clear intent of this is that performance can be gained, as the application parts do not need to reside on the same physical component of the readout unit. In the ideal case, Input and Output control can be performed on two separate processing units and data transfer through independent communication buses.

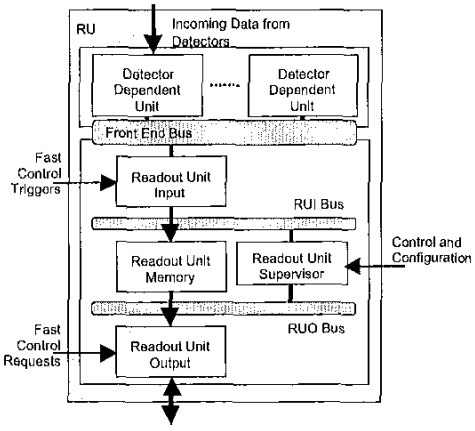


Figure 2: Readout unit functional decomposition

IV. Software Components

A. The Approach

With object-oriented programming languages becoming ubiquitous, we are able to address the above-mentioned requirements. By encapsulating differences in accessing communication channels into specialised software components, we are able to satisfy two demands at the same time: (i) high performance by optimised access to particular devices and (ii) a complete abstraction of communication by having a well defined interface for different kinds of data transport layers. In general, software components comprise closely collaborating objects and present functionality through a clean interface to the outside world [11],[12]. Using this approach we can connect components through which data streams flow. On top of it, richer paradigms can be placed. Figure 3 shows a layered view of the RU software.

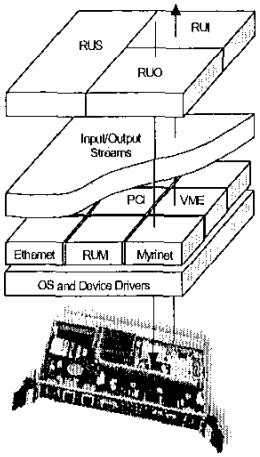


Figure 3: Software layout of the CMS readout unit

The top layer contains the application components in which the actual functions of the Readout-Unit are implemented. It is based on an I/O streams package. The latter one de-couples the application from the system specific functions and provides a high-level interface to data communication devices. The significant advantage is that, we can accomplish the major goal of having a completely uniform interface between the calling program and the underlying hardware.

B. Readout Unit Components

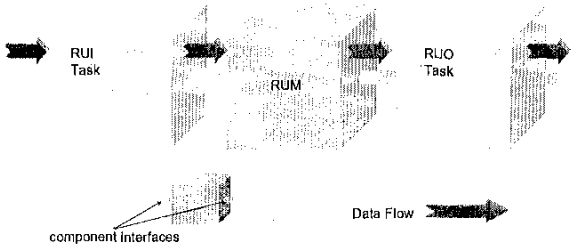


Figure 4: Software components of the readout unit

In our problem space, there are a number of logical activities that occur at the same time. For example, the RUI continuously reads data from the detector front-end and may use asynchronous hardware interrupts to request services. In addition, the RUO services requests from the filter units and control several independent devices such the readout unit memory and the interface to the switch. As outlined above, what we desire is the ability to represent these parallel activities and model the RUI and the RUO as two

independent software tasks. Consequently, RUI and RUO are *active* components and exchange information through a third element, the RUM, as Figure 4 illustrates [10]. The RUI and the RUO are *composite* objects. They can be seen as an aggregate of *input/output streams* [13],[14] used for control and data transfer.

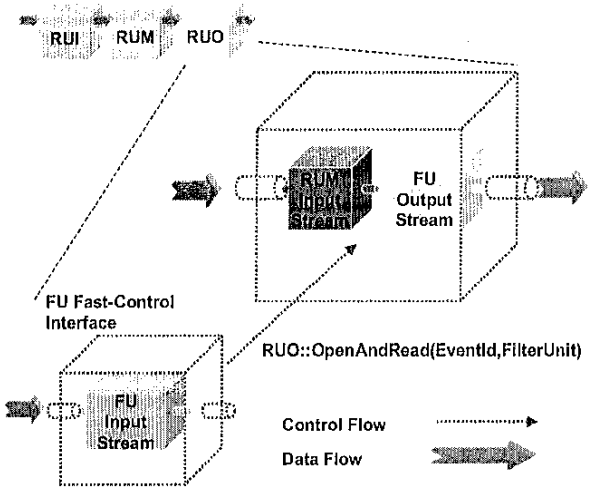


Figure 5: Data flow and fast-control flow in RUO

As shown in Figure 5, the RUO is essentially made of two basic components. The first component receives requests through an input stream from the filter units and translates them into specific control actions. The second component consists of an input stream used for accessing the RUM and an output stream used to distribute event fragments to the requesting filter unit. The dashed lines express the *thread of execution* of the requested action from the filter unit. The I/O streams for this composite object can be prepared at configuration time to operate on the available hardware or software support. For example, the control stream can be configured on Ethernet, while the RUM stream can be configured to access a CPU board memory pool. The FU output stream can be configured for a Myrinet [15] switch interface. Similarly, the RUI consists of an input stream for receiving readout requests (triggers), an output stream to the RUM and an input stream to access VME memory.

C. VME – RU Memory Transfer Coding Example

This example depicts how data can be transferred from VME to a dual-ported memory board and vice versa by using I/O streams components. First the input and output streams have to be configured. This can be done in a generic way, so that the same kind of stream can be used for different communication devices. The example reflects the task performed by the RUI.

```
vxiostream iStream  ( ...VME config ...);
vxiostream oStream  ( ...RUM config...);
```

In order to make use of the DMA engine of the VME board, this component has to drive the data transfer action. A manipulator (`setl`) can be used to determine how much data has to be transferred. Streams can be tied together directly:

```
oStream.open(...Event fragment id...);
iStream >> setl(length) >> oStream;
oStream.close();
```

Similarly, if data should be pushed into the dual port memory using a copy operation, the streams are used as follows:

```
oStream << setl(length) << iStream;
```

D. Dual Port Memory to Network Coding Example

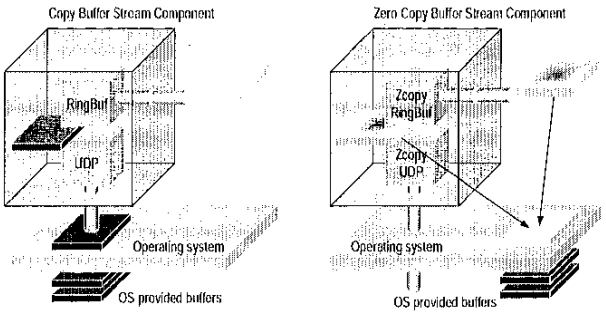
If data is moved from the RUM to a network card, as applied in the RUO shown in Figure 5, the following sequence is used:

```
vxiostream iStream ( ...memory config... );
vxiostream oStream ( ...net config... );
vxiostream ctrlStream (...TCP config... );

RUO::svc()//execute in a task context
{
   Loop Forever
   {
     ctrlStream >> setl(sizeof(Request)) >> request;
     this->OpenAndRead(request);
   }
}

RUO::OpenAndRead(Request & request)
{
   iStream.open(request.EventFragment());
   oStream.open(request.EventFilter());
   oStream << setl(length) << iStream;
   iStream.close();
   oStream.close();
}
```

In the given case, the network card component will drive the data transfer. As such network cards are equipped with a DMA engine, the component makes use of it. Although we would like to follow the requirement to design symmetric components [6] using the stream operator in both directions does not always provide us with the best performance. If we can use a DMA engine that performs the data copying, we offer the application programmer only one possibility. E.g. in the example above, the instruction

```
istream >> setl(length) >> ostream;
```

is not implemented, because `istream` is not armed with a DMA engine. This restricts the number of possible use cases, but eases the implementation of highly efficient programs.



E. Zero Copy Streams

Figure 6: Components encapsulate various different tasks, but expose the same clean and narrow interface.

The implementation of the I/O streams aim at achieving good performance. Therefore they rely on zero-copy buffers wherever possible [16],[17],[18]. In general, driver interfaces are characterised by copy semantics, where input and output operations *transfer data* between the kernel and user-defined buffers. Although simple, this behaviour limits the ability of the operating system to efficiently perform data transfer. Instead of copying data between kernel and user space, the system will *pass addresses* of pre-allocated buffers, as shown in Figure 6 so that zero-copy transfer can be offered. So far I/O streams were seen as black boxes. This permitted us to concentrate on the way they can be connected for data exchange. Let us briefly mention how they are configured. There is one major structural element which is common to all streams and which is relevant to the effectiveness of the I/O stream paradigm. Interprocess communication components are used to configure the streams for different hardware and software interfaces. Such an element can be seen as an adapter for a given hardware and is prepared at configuration time. An adapter is assigned to each stream at instantiation time. This is shown in the examples as configuration. Streams benefit from adapters in order to perform their tasks, see nested components in Figure 6.

V. System Evaluation

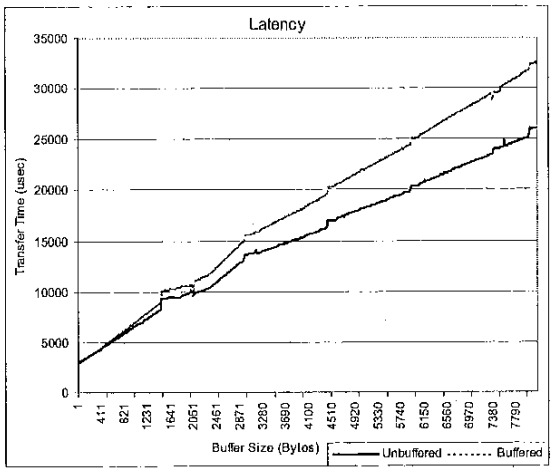


Figure 7: UDP roundtrip measurement on 100 Mbit Ethernet. The differing slopes indicate the effect data copy operations.

The main characteristics that must be provided by the software system once it is applied in a real data acquisition environment are reliable operation and deterministic performance. For these reasons we chose the VxWorks RTOS [19]. The hardware used to perform the development was a Motorola PPC MV2034 VME-based controller [20]. Standard 100 Mbps Ethernet was used for control purposes and Myrinet [15],[21] served for data transfer between readout and processing units. The RUM has been implemented using a custom board to allow storing and retrieving memory fragments in a file system like manner.

A. Performance of Zero Copy Streams

A comparison between UDP transfer based on the ordinary socket interface and using an optimised protocol stack with zero copy buffers has been performed [22]. Although no modifications in the structure of the application program had to be performed, performance could be increased significantly. From Figure 7 we see, that the effect becomes more visible with increasing buffer sizes. The traditional component has to copy data several times in order to present the application programmer with byte-stream semantics.

B. Overhead of the Software Components

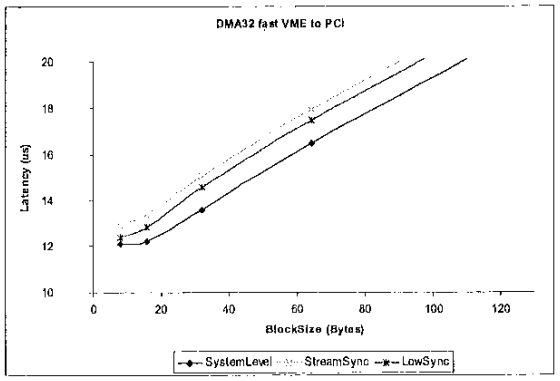


Figure 8: Latency measurements for DMA 32bits VME to PCI transfers using three different software tools

Figure 8 shows the latency slopes measured for DMA transfers from VME to PCI. We evaluated three different software approaches. The highest performance is obtained with raw system access. This line serves as a reference to exhibit the overhead induced by additional software layers. In this particular case the overhead measured for the I/O streams implementation compared to the system level is about 1 µsec. Some better performance can be seen with a higher level operating system call that hides to some extent the differences in the hardware, but does not provide the flexibility offered by the stream abstraction. As the differences of the slopes are constant for all block sizes, the behaviour of the streams software is deterministic.

VI. Related Work

For real-time systems only few people investigated configurable software components. Research in object

oriented RT operating systems has been done by the Chorus group [23],[24], resulting in a commercial product distributed by Sun Microsystems.

We are also investigating another approach like the ADAPTIVE Communication Environment (ACE) [25],[26]. This object oriented programming toolkit provides an abstraction layer on which components for network programming are based. It is used along with a real-time CORBA implementation in avionics environments [27].

The goal of all these efforts is to obtain a treasure of reusable components, so that systems do not have to be built from scratch each time the platform is changed or new application functionality is introduced. This is common practice in desktop computing and should also become the everyday working manner for real-time and embedded systems. Although the related work shows that research groups go this direction, problems such as efficiency and predictability are not yet entirely solved.

VII. CONCLUSIONS

We have presented how object oriented techniques can be applied to provide a flexible and efficient software architectures for data acquisition systems. This design pattern is extremely well suited for tackling problems related to subsystem connection in highly diverse environments and providing an infrastructure for rapid prototype development.

We have also shown some of the advantages over traditional systems programming approaches, such as increased configurability by having well encapsulated behaviour of the hardware and good performance by offering highly optimised components for each kind of communication hardware. In addition, the approach provides easier software development due to the homogeneous view of the underlying hardware. We experienced this when applying the proposed approach in an 8 by 8 demonstrator DAQ system [28].

The work will evolve to cover various other technologies such as PCI and IOP enabled architectures. Components that follow the presented scheme will be available for these technologies such that we can reuse our applications without modifications. Success in performing this task would further prepare the ground for component-based software engineering in high-performance data acquisition systems.

VIII. REFERENCES

[1] A. Kruse, CMS Online Event Filter Software, *Computer Physics Communications* **110,** Elsevier Science Pub., 1998 pp. 102-106

[2] D. Batory and S. O'Malley, The Design and Implementation of Hierarchical Software Systems with Reusable Components, *ACM Transactions on Software Engineering and Methodology*, 1(4), Oct 1992, pp. 355-398.

[3] D. McIlroy, Mass-produced software components, in: P. Naur et al. (eds.), Software Engineering Concepts and Techniques (in Petrocelli/Charter, 1976, pp. 88-98)

[4] D. L. Parnas, Designing Software for Ease of Extension and Contraction, *in IEEE Transactions on Software Engineering*, **SE-5**(2), March 1979, pp. 128-137

[5] O. Nierstrasz, S. Gibbs and D. Tsichritzis, Component-Oriented Software *Development Communications of the ACM* **35**(9): 160-164, September 1992

[6] A. N. Habermann, L. Flon and L. Cooprider, Modularization and hierarchy in a family of operating systems, *Communications of the ACM*, **19**(5), May 1976.

[7] The CMS Collaboration, The Compact Muon Solenoid, CERN, *Technical Proposal*, No. 7, LHCC 94-38 December 1995.

[8] C.G. Masi, Software smartens up Data Manipulation, *IEEE Spectrum*, April 1999, pp. 30-35.

[9] I. Foster, J. Geisler, C. Kesselman, S. Tuecke, Managing Mul-tiple Communication Methods in High-Performance Networked Computing Systems, *J. Par Dist Comp* **40**(1): 35-48, 1997.

[10] S. Cittolin, A. Fucci, P. Sphicas and K. Sumorok, *Dual Port Memories in LHC Experiments*, CERN, CMS-RD12 TN 95-04, March 1995.

[11] C. Pfister and C. Szyperski, Why Objects Are Not Enough, in: Proceedings of the First International Component Users Conference (SIGS Publishers), Munich, Germany, July 1996.

[12] I.Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure (Morgan Kaufmann Pub., Inc. San Francisco,CA,USA) 1999, pp 259-278.

[13] D. Ritchie, A Stream Input-Output System. AT&T Bell Labs Technical Journal **63**(8): 311-324, October 1984.

[14] K. Kreft and A. Langer, Deriving from IOStreams, C++ Report, SIGS Publisher, September 1995.

[15] N. J. Boden, D. Cohen an R. E. Feldmann, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W-K. Su: MYRINET: A Gigabit per Second Local Area Network, IEEE Micro **15**(1):19-35, Feb 95.

[16] M. Lauria and A. Chien, MPI-FM: High Performance MPI on Workstation Clusters, *J Par Dist Comp* **40**(1): 4-18, Jan 1997.

[17] T. von Eicken, A. Basu, V. Buuch and W. Vogels, U-Net: A User-Level Network Interface for Parallel and Distributed Computing, in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 40-53, ACM Press, December 3-6, Copper Mountain Resort, Colorado, USA, 1995.

[18] F. O'Carroll, H. Tezuka, A. Hori and Y. Ishikawa, The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network, in Proceedings of the *1998 International Conference on Supercomputing* (ICS 98), pages 243-250, ACM Press, Melbourne, Australia, 1998.

[19] Wind River Systems, Corporate Head Quarters, 500 Wind River May, Alameda, CA 94501.

[20] Motorola Computer Group, Corporate Head Quarters, 2900 S. Diable Wy, Tempe, AZ 85281.

[21] Myricom, Inc. High-Speed Computers and Comm., 325 N. Santa Anita Ave. Arcadia, CA 91006.

[22] WindRiver Systems, *SENS for Tornado Component Release Supplement*, 1.0, Ed. 1, January 1998, DOC-12286-ZD-02

[23] S. Habert, L. Mosseri and V. Abrossimov, COOL: Kernel Support for Object-Oriented Environments, *Proc. of the Joint ECOOP/OOPSLA '90 Conference*, Oct. 1990, Ottawa, Canada

[24] M. Guillermont – Chorus/ClassiX r3 Technical Overview. *Chorus Systems Technical Report* CS/TR-96-119.13, May 1997.

[25] D.C.Schmidt, The ADAPTIVE Communication Environment. An Object Oriented Network Programming Toolkit for Developing Communication Software, 12th Sun User Group Conference, San Jose, CA, USA, December 7-9, 1993.

[26] D. C. Schmidt, T. Harrison and E. Al-Shaer, Object-Oriented Components for High-speed Network Programming, *Proceedings of the Conference on Object-Oriented Technologies*, USENIX, June, 1995, Monterey, USA, pp. 21-38.

[27] F. Kuhns, D. C. Schmidt, D. Levine and R. Bector, The Design and Performance of RIO – a Real-Time I/O Subsystem for ORB Endsystems, *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium* (RTAS99), Vancouver, British Columbia, Canada, June 2-4, 1999.

[28] The CMS Event Builder Demonstrator based on Myrinet, F. Meijers, et al, *same conference*