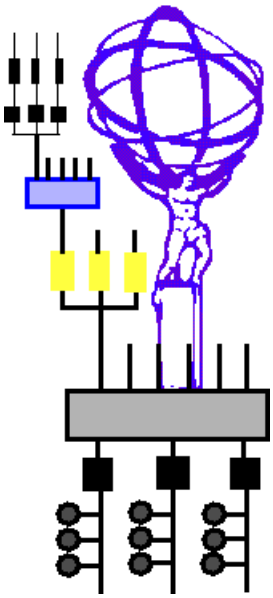
ATLAS DAQ Note ATL-DAQ-2000-005

Event Filter Summary Document

ATLASTrigger-DAQ

Event Filter Summary Document

Document Version:	1
Document Issue:	1
Document Status:	Final
Document Date:	March 6, 2000



European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

Abstract

A summary of the work performed by the ATLAS Event Filter group is presented. This work has been done in the context of the DAQ / EF-1 project, in preparation of the DAQ, High Level Triggers and DCS Technical Proposal. The following people have been involved in the present work:

ALBERTA: R. Davis, P. Green, J. MacKinnon, J. Pinfold

BARCELONA: M. Bosman, A. Pacheco

BERN: H.P. Beck, R. Mommsen, A. Radu

CERN / EP: D. Francis, R. Jones, L. Mapelli, G. Mornacchi

CERN / IT: C. Boissat, F. Hemmer, C. Rafflin

MARSEILLE: C. Bee, P-Y. Duval, F. Etienne, E. Fede, D. Laugier, C. Meessen, R. Nacasch, Z. Qian, F. Touchard

PAVIA: A. Negri, G. Polesello, D.A. Scannicchio, V. Vercesi

ROMA: C. Stanescu

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	ATLASTrigger-DAQ Event Filter Summary Document		
	Version:	1		
	Issue:	1		
	Edition:			
	ID:			
	Status:	Final		
	Created:	October 27, 1999		
	Date:	March 6, 2000		
	Access: :			
	Keywords:			
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	5.5

Table 1 Document Control Sheet

	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**			
	Written by:			
	Reviewed by:			
	Approved by:			

Document Status Sheet

Table 2 Document Status Sheet

Title:	ATLASTrigger-DAQ Event Filter Summary Document		
ID:			
Version	**Issue**	**Date**	**Reason for change**
0	0	27/10/1999	Document creation
1	0	18/02/2000	First official release
1	1	06/03/2000	Typo corrections

Document Change Record

Table 3 Document Change Record (of changes made since version ...)

Title:	ATLASTrigger-DAQ Event Filter Summary Document		
ID:			
Version: 1.0]		**Originator:** F. Touchard	
Date: February 18, 2000		**Approved By:**	
Page	**Paragraph**	**Reason for Change**	
		Document creation	

Table of Contents

Abstract 3

Document Control Sheet 3

Document Status Sheet 5

Document Change Record 5

Table of Contents 7

Chapter 1
Introduction 1

Chapter 2
Architecture 3
2.1 Introduction 3
2.2 Global view 3
2.3 Sub-farm architecture 4
2.4 Event Handler architecture 5
2.4.1 Introduction 5
2.4.2 User requirements 6
2.4.2.1 Data flow - EH interface 6
2.4.2.2 Distributor 7
2.4.2.3 Processing Task 7
2.4.2.4 Event Handler Supervisor 7
2.4.3 Data flow components 7
2.4.3.1 Distributor 7
2.4.3.2 Collector 8
2.4.3.3 Functional model of the Event Handler 9
2.4.4 Supervision 9
2.4.4.1 Supervisor functionalities 10
2.4.4.2 Event Handler Run Control, states and transitions 11
2.4.4.2.1 EH States 11
2.4.4.2.2 EH Transitions 12
2.5 References 14

Chapter 3
Software detailed design and generic implementation 15
3.1 Data flow through the Event Handler 15
3.1.1 User requirements 15
3.1.1.1 Event transfer between component and with SFI-SFO: 15

3.1.1.2 Component control 16
3.1.1.3 Component behaviour 16
3.1.1.4 The Event Filter 16
3.1.2 Component design 16
3.1.3 Element states and methods 19
3.1.3.1 States 19
3.1.3.2 Element methods 19
3.1.3.2.1 Core Element 19
3.1.3.2.2 IO Elements 21
3.1.3.2.3 Control Element 22
3.1.4 Component implementations 22
3.2 Event Handler API 23
3.2.1 Introduction 23
3.2.2 SFI - Distributor functions 23
3.2.3 Collector - SFO functions 23
3.2.4 Example of timing chart 24
3.3 Supervision 25
3.4 References 25

Chapter 4
Commodity PC prototype 27
4.1 Introduction 27
4.2 The available testbeds 27
4.2.1 The Marseille farm 27
4.2.2 PCSF 28
4.2.3 EFF 28
4.3 The data flow implementation 29
4.3.1 ILU protocol 29
4.3.2 TCP protocol 30
4.4 The supervision implementation 30
4.4.1 JAVA Mobile Agents 31
4.4.2 Functionalities of the Supervisor 33
4.4.3 Other possible technologies 33
4.5 Performance 34
4.5.1 Conditions of the tests 34
4.5.2 Communication protocol 35
4.5.3 Throughput 35
4.5.4 Scalability 36
4.5.5 Robustness 36
4.6 Integration with the DAQ/EF-1 prototype 36
4.7 Conclusions and outlook 37
4.8 References 37

Chapter 5

Symmetric Multi Processor prototype . . . 39
5.1 Introduction . . . 39
5.2 Sub-farm implementation . . . 39
5.2.1 Dataflow . . . 40
5.2.2 Supervision . . . 41
5.2.3 Error handling . . . 41
5.3 Sub-farm performances . . . 42
5.3.1 Hardware . . . 43
5.3.2 Throughput . . . 43
5.3.3 Load balancing . . . 44
5.3.4 Scalability . . . 45
5.3.5 Robustness . . . 46
5.4 Integration with the DAQ/EF-1 prototype . . . 47
5.5 Conclusions . . . 47
5.6 References . . . 48

Chapter 6
INTEL Commodity Multi-Processor Approach . . . 49
6.1 Event Filter Implementation . . . 50
6.2 Architecture . . . 50
6.3 Operating System . . . 51
6.4 Communications Protocol . . . 52
6.5 Details of the Sub-Farm . . . 53
6.6 Integration . . . 54
6.7 System Tests . . . 54
6.8 Future Plans: . . . 56
6.9 References . . . 56

Chapter 7
Comparison of the prototypes . . . 57
7.1 Data redundancy and robustness . . . 57
7.2 Data communication mechanisms . . . 57
7.3 Throughput and scalability . . . 58
7.4 Sub-farm configuration and monitoring . . . 58
7.5 Architectures and costs . . . 59

Chapter 8
Outlook . . . 61

Chapter 1
Introduction

This document summarises the work which has been done in the Event Filter group of the DAQ/EF-1 Prototype project in the last three years. The interface between the Event Filter and the other sub-systems of the DAQ/EF-1 system have been very precisely defined so as to facilitate parallel development, given that all the effort on the Event Filter comes from outside CERN.

It was realised early on that there were several possible candidate hardware architectures for an event filter computing farm (EFF) capable of treating an input data rate of ~1 GB/s (1 kHz) and effecting a factor of ten reduction of this rate by applying complex physics constraints to the events. The required processing power was estimated to be at least 25 kSPECint95, or 1 TIPS. Any final hardware choice must clearly be left until the latest possible moment compatible with building and commissioning the EFF in time. It was also noted that given the estimated running lifetime of >15 years, the farm must be both easily and gradually upgradeable. This indicates the possibility that a variety of hardware and even operating systems may be running in parallel during the lifetime of the EFF.

With this in mind, the group has produced a high level design of the EFF software architecture which is deliberately independent of any specific hardware or O/S features. This design is presented in Chapter 2. We have implemented this design in C++ in a modular fashion, and giving specific attention to isolating any hardware or O/S specific areas, thereby leaving the body of the code invariant under hardware or O/S changes. This implementation is described in Chapter 3. This chapter also describes the interface of the EFF dataflow software with other elements of the DAQ/EF-1.

Three EFF prototypes have been designed and constructed based on three somewhat different hardware; Commodity PCs, Commercial SMPs and INTEL Commodity MultiProcessors. The design described in Chapter 2 was implemented in each case while capitalising on the specific features of each prototype. The common software (Chapter 3) has already been implemented on the Commodity PC prototype, and will be in the near future on the other two. Chapters 4, 5 and 6 describe in detail the implementation of the three prototypes and the measurements and tests made on each. We have made a big effort to coordinate the tests and measurements in order to make some cross-prototype comparisons. These comparisons are presented in Chapter 7 and the document concludes with an outlook for future work on the EFF towards the T/DAQ TDR in June 2001, in Chapter 8.

Chapter 2
Architecture

2.1 Introduction

The main objective of the Event Filter (EF) is to reduce the rate of data sent to mass storage by a factor ten relative to the input rate from the LVL2 trigger. This should be achieved by applying to the fully assembled events (downstream of the Event Builder) the complex algorithms from the offline reconstruction and subsequent physics analyses. In addition to this primary goal, the EF will tag events of special interest so that they are quickly analysed in detail. Because the EF is the first location in the dataflow where fully assembled events are available, it is ideally suited to the online monitoring of the whole detector and of the physics (trigger efficiency) quality. Calibration and alignments tasks will also be done at the EF level since they are crucial to the filtering quality and should therefore be performed as soon as possible.

Data security must be ensured while processing events in the sub-farm. The design goal is that neither hardware nor software problems should lead to the loss of an event. Reliability and robustness must be implemented at all levels. In particular, special attention must be devoted to the reconstruction and filtering software quality.

2.2 Global view

The DAQ system and the Event Filter are interfaced based on the following assumptions:

- the Event Builder receives fragments from the ReadOut Buffers and assembles them into fully built events
- the events are passed to the Event Filter through multiple independent ports
- after processing, the events are collected by the data flow to be sent to permanent storage

No assumption is made on the technology which is used for both the event building (switch, shared memory, etc.) and the mass storage (single channel, multi channel, database, etc.)

The Event Filter is therefore factorised into independent *sub-farms*, each of which is connected to a different output port of the Event Builder. The required number of sub-farms will be determined from the available technology and the total CPU power needed.

Every sub-farm contains one or several processors linked together. The design will be independent of the technology of the processors as well as of the communication protocol between these processors (bus, crossbar switch, local or wide area network, etc.).

A sketch of the global organisation of the Event filter can be seen in Figure 1.

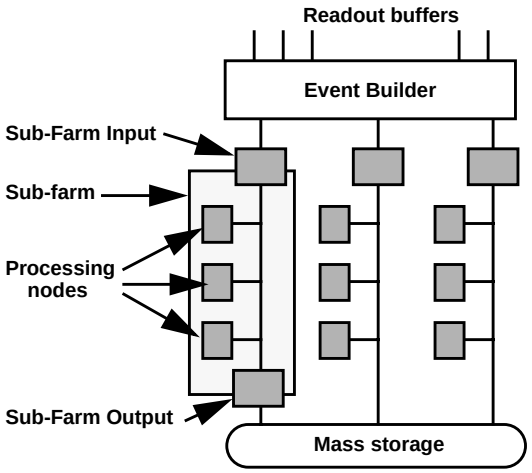


Figure 1 Global view of the Event Filter architecture

2.3 Sub-farm architecture

Every sub-farm of the Event Filter is logically divided into three basic components which are displayed on Figure 2:

- the Event Handler (EH) which processes the events
- the sub-farm DAQ unit (SFD) which is responsible for channelling all event data coming from the event builder through the EH. Events transformed by the EH are then passed by the DAQ to the mass storage system
- the LDAQ which provides the means to control the flow of data in the SFD and makes the interface with the Back-End software

The SFD and the LDAQ are the responsibility of the Dataflow group [1] and are described elsewhere.

The EH will implement the filtering capabilities as well as other processing functionalities (such as monitoring, calibration, etc.). A supervisor implementing the control and local monitoring capabilities of the sub-farm will be part of the EH.

On the data flow side, the interfaces with the EH are provided by the Sub Farm Input (SFI) and Sub Farm Output (SFO) which are instances of a dataflow I/O module. APIs are implemented by the EH in the so-called *distributor* and *collector* logical modules (Figure 2). They will be described in Chapter 3 .

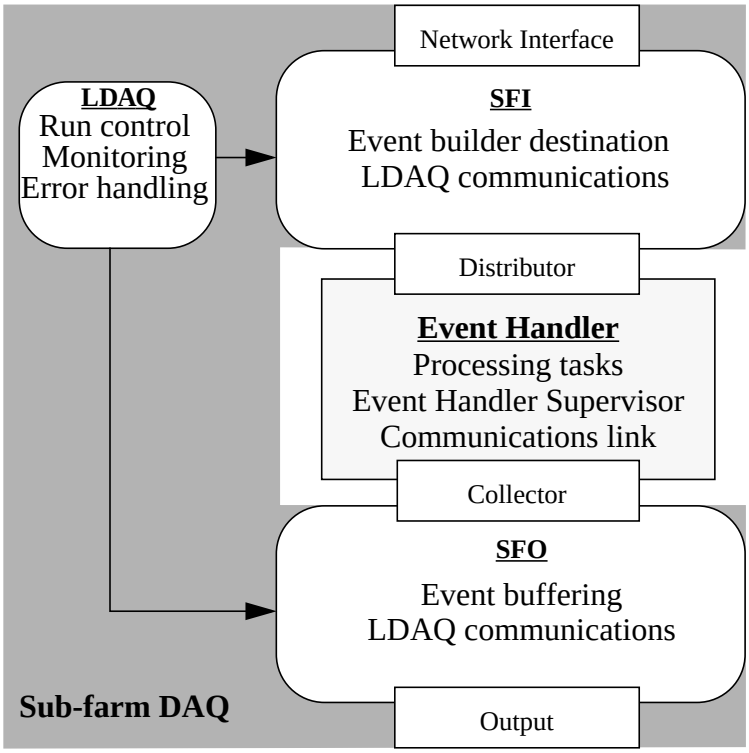


Figure 2 Sub-farm architecture

2.4 Event Handler architecture

2.4.1 Introduction

This section summarises the content of Technical Note 61 [2].

As shown in Figure 3, the Event Handler consists of

- a *distributor* responsible for dispatching events to the different processing elements and providing some capacities to choose the destinations according to parameters available in the event header
- one or several *processing elements*, comprising processing tasks
- a *collector* gathering processed accepted events before making them available to the data flow
- a *supervisor* providing the control and monitoring capabilities

The first three elements are the *data flow* elements of the EH. The flow of events through the EH follows the path: *Distributor* → *Processing Element* → *Collector.*

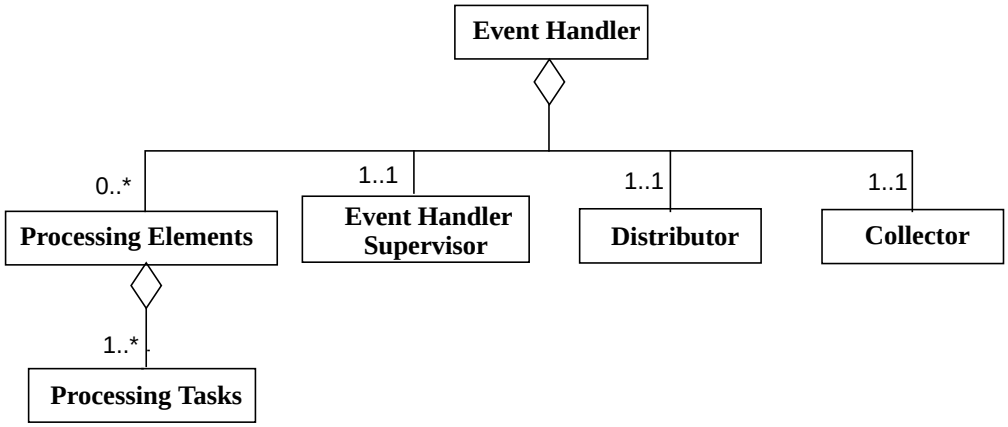


Figure 3 Event Handler object model

2.4.2 User requirements

We list here the main user requirements for the different objects constituting the EH. The first item (referred as Data flow - EH interface) is common to the Distributor and the Collector.

2.4.2.1 Data flow - EH interface

1. The interface with the DataFlow *shall* be independent of different Event Handler architectures.
2. The interface with the DataFlow *shall* allow the SFI and SFO to function without processing tasks.
3. An event *shall* not remain in an SFI buffer during processing by one or more event handler processing tasks.
4. The interface with the DataFlow *shall* be independent of the details of the event format and of the mechanism of event transfer.
5. The Event Handler *should* ensure that events are recoverable in case of a run-time error.
6. The details of the internal structure of the Event Handler *shall* be described in a configuration database.

2.4.2.2 Distributor

1. The Distributor *shall* have the capacity to selectively distribute events according to parameters available in the event header.

2.4.2.3 Processing Task

1. A processing task *shall* have a unique identifier within a given Event Handler
2. A processing task *shall* be associated to a single event type

2.4.2.4 Event Handler Supervisor

1. The Event Handler *shall* have an Event Handler supervisor.
2. The Event Handler supervisor *shall* provide the interface with the Back-End system and extend the functionality of the Back-End system into the Event Handler.
3. The Event Handler Supervisor *shall* provide process management functionality within its Event Handler.

2.4.3 Data flow components

Because of the architecture choice of the Main Data Flow, events are pushed from the SFI to the EH. The flow of events through the EH is then purely data driven. The processed events are pulled from the Collector by the SFO. The regulation of the flow of events is naturally done by back pressure based on availability of resources.

2.4.3.1 Distributor

The Distributor receives events from the Sub-Farm DAQ and forwards them to the processing tasks within its Event Handler. In addition, the Distributor ensures that events are not lost during the time that events are treated by the Event Handler.

The treatment of events according to event type is an option which is made available in the design. It may well prove desirable to have the ability to filter events of differing type (e.g. physics, calibration, monitoring) to different specific processing tasks. A *pull* protocol between the processing tasks and the distributor is thought to give the best functionality since the blocking of a processing stream should not precipitate the blocking, by starvation, of the others.

Events are immediately buffered in a global (to the sub-farm) data store (Distributor Global Buffer - DGB) on reception from the SFI. This buffer is associated to a permanent storage device so that data security is ensured all along the lifetime of the event inside the sub-farm. When the event has been either accepted and successfully transferred to the downstream component, or rejected, it is removed from the DGB. On the occurrence of a problem while being processed, given the fact that the event is still in the DGB, a recovery procedure can be launched according to the origin of the problem (hardware or software problem, run time or

system error, etc.). Implementation scenarios of the DGB function, to provide the best trade-off between security and efficiency will have to be studied in detail in the future.

A simplified view of the Distributor object model is given in Figure 4.

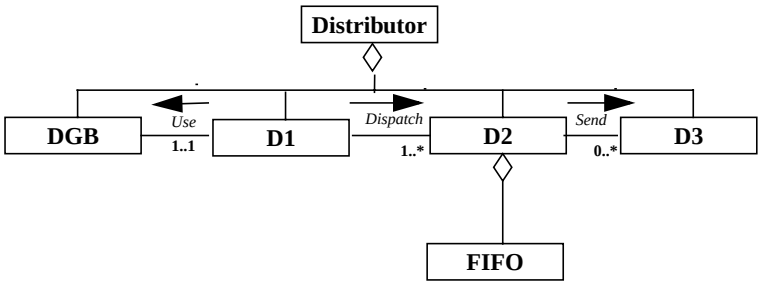


Figure 4 Object model of the Distributor

D1 implements the EH API to receive events from the SFI and is responsible for the event queue management. According to their type (contained in the event header), events are sent to the appropriate instance of the D2 component. The D3 component acts as a buffer for the processing tasks and requests events from the D2 component as soon as they have free space for a new event. The separation of D2 and D3 is made in anticipation of the distributed nature of the EH. It is possible to collapse the two components into a single one depending on the characteristics of the processing machines (e.g. a SMP machine) or of the communication protocol (e.g. CORBA).

2.4.3.2 Collector

All events which have been selected by the processing task for permanent storage are made available to the SFO independently of their type. The object model of the Collector is symmetric to the Distributor one, but somewhat simpler since no component is necessary for a sorting function.

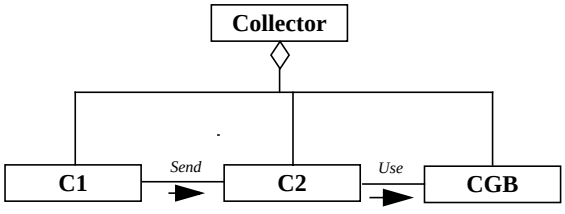


Figure 5 Object model of the Collector

C1 (the “Event Collector”), of which there will be an instance per processing task, will provide the processing task with the means of disposing of the event after processing (*OutEvent*).

C2 (the “Event Receiver”), of which there will be one instance per sub-farm, will ‘receive’ events sent by the C1s and store them in the Collector Global Buffer (CGB) which plays a role similar to the one played by the DGB in the Distributor. The two buffers have been separated to handle in a simpler way any extra information which may have been added by the processing task. C2 is responsible for the buffer management of the CGB. It is also required to inform the appropriate C1 on the successful completion of the event transfer to the CGB.

2.4.3.3 Functional model of the Event Handler

Figure 6 shows a functional model of the EH when two different event types are to be taken into account. It is recalled that the DGB and the CGB are not mandatory, but provide data security. The choice to implement them will be a trade-off between the security which is brought and the slowing down of event handling which is introduced.

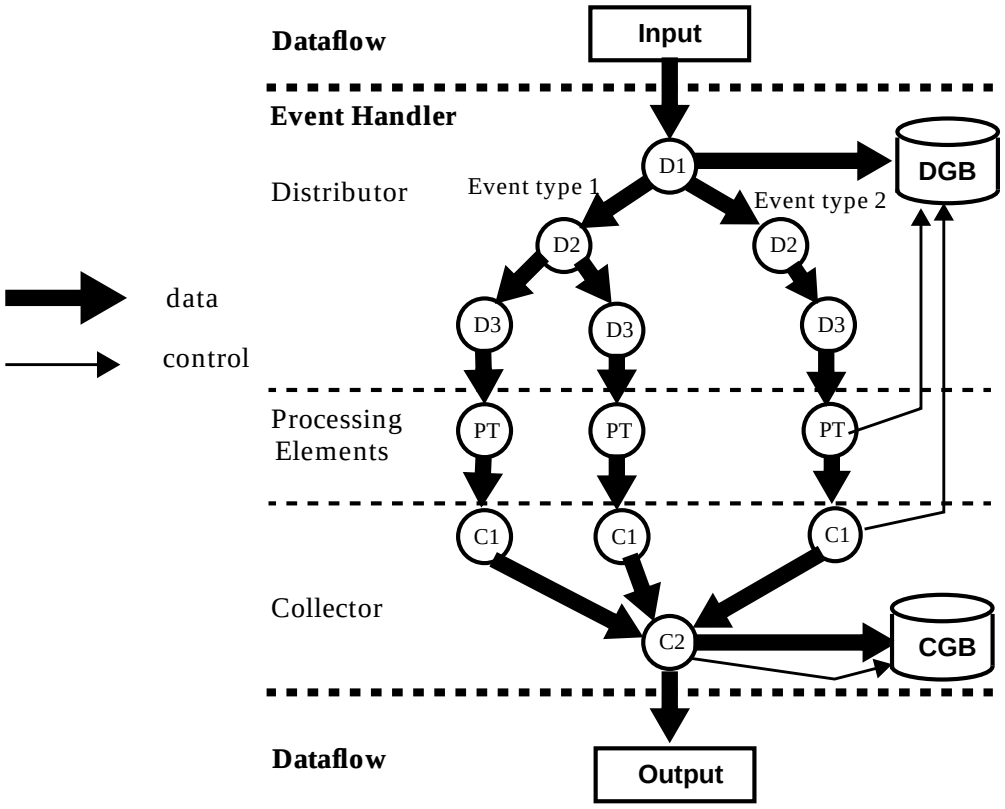


Figure 6 Functional model of the Event Handler

2.4.4 Supervision

This section summarises the content of Technical Note 92 [4].

2.4.4.1 Supervisor functionalities

The functions which must be fulfilled by the EH Supervisor are:

- run control: it provides the interface between the EH and the overall Run Control system. A local standalone control system must allow some local direct actions on the EH without affecting the functioning of other parts of the Event Filter (e.g. elements of the sub-farm DAQ: SFI, SFO)
- process management: it must be able to launch and stop the different EH tasks. It should also be able to detect process crashes and take appropriate action
- access to databases: it is responsible for providing access to the processing tasks to up-to-date databases for calibration and alignment. It should also inform the other EH components of any conditions and general purpose variables (e.g. the run number) which drive their behaviour, as well as accessing the local EH configuration database for process management purposes
- monitoring: information on the EH behaviour must be collected and published. Errors should be reported to the Backend error facility (MRS [3]). It should also be able to collect and publish information provided by the processing tasks.

The Supervisor functions are performed via several types of devices (see Figure 7):

- a command user interface used to receive commands from an operator in local mode
- a set of specific functions to handle process management
- a set of monitoring displays to present the EH states and variable values to users
- an external dynamic database accessed:
  - to publish EH states and variables which have to be made available to the external world
  - to read external DAQ system states and variables which may be of concern to the EH activities.
- a message system to report asynchronously to the end users

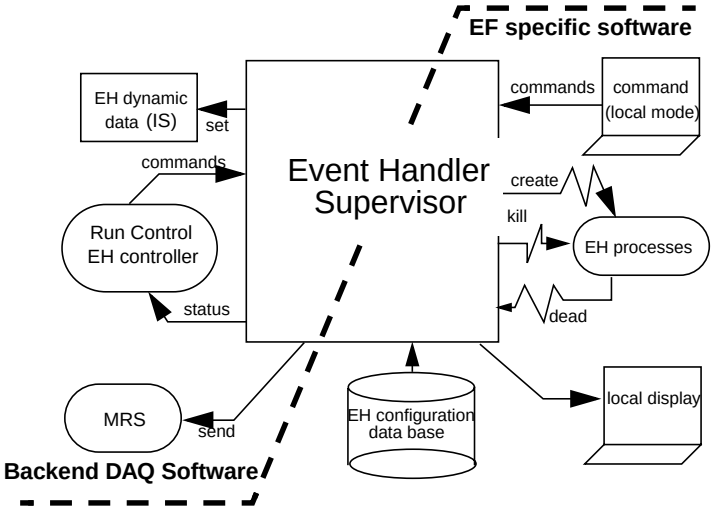


Figure 7 The EH supervisor context diagram

Minimization of the synchronization and coupling between the event flow and the supervisor has been one of our first concerns in designing the EH [2]. The flow of the events through the EH is purely data driven and does not rely on transactions with the Supervisor.

2.4.4.2 Event Handler Run Control, states and transitions

This section details the states and transitions which describe the global EH behaviour as it is seen by the user or the DAQ run control system. Before doing so however, some explanation of the behaviour of the internal EH elements is required.

The Distributor, the Processing Tasks and the Collector are objects the states of which do not map a hierarchical finite state machine (HFSM). The states or properties of the objects are known of the Supervisor and are used to build the overall state of the EH.

Three states characterize the Distributor and Collector from the Supervisor's point of view, namely: some events are available for treatment (*not-empty*); no event is available for treatment (*empty);* the task does not exist (*absent)*. Similarly, for Processing Tasks: processing an event (*processing*); waiting for an event (*idle*); the task does not exist (*absent*).

The EH Hierarchical State Machine is shown in Figure 8 which also displays the global 'superstates': *Idle, Working, Active* which are used in the control logic of the HFSM itself.

2.4.4.2.1 EH States

Initialized

The EH Supervisor has been started and is ready to accept external commands.

Loaded

The Supervisor knows the configuration. All the necessary EH components Distributor, Collector and a minimum number of Processing Tasks have been created.

Configured

All the necessary EH components Distributor, Collector and a minimum number of Processing Tasks have been checked ready to process events. The EH is in the superstate *Working*.

Running

The EH is processing events

Paused

The EH data input has been paused

Dead

All tasks including the Supervisor itself have been killed

A second state machine runs in parallel in the superstate *Alive*. This FSM allows to deal with errors either when executing a requested transition or when an unsolicited change in the state of a component occurs. A transition between the *OK* and *Bad* states is then executed.

2.4.4.2.2 EH Transitions

After having executed any operation, a status code is reported, otherwise stated.

load: Initialized → Loaded

The EH Supervisor reads from the configuration database the selected DAQ configuration and the corresponding EH configuration file. All the EH components are created.

unload: Any → Initial

Kill all EH components with the exception of the Supervisor and release configuration DB resources if any.

configure: Loaded → Configured

All the EH components are checked ready to process events (communication links have been opened, calibration and geometry databases have been read by every processing task, etc.).

unconfigure: Configured → Loaded

Blank operation.

Terminate: Idle → Dead

Kill all EH components including the Supervisor.

kill: Idle → Dead

Kill all EH components including the Supervisor. No status code is reported.

start: *Configured → Running*

The Supervisor broadcasts the latest run number (*current run number*) and the calibration version (*current calibration*) to all the processing task environments. If a new version of the calibration[1] is available, it must ensure that this is made available to each processing task before event processing commences.

The SFI begins sending events to the Distributor, and the EH immediately reaches the *Running* state. Some specific actions may be performed by the processing tasks before they read events from the Distributor FIFOs.

stop: *Running → Configured*

The Supervisor asynchronously informs the EH tasks of the occurrence of the command. The SFI stops sending events to the Distributor. The Configured state is reached when the Distributor and Collector are *empty* - or *absent* - and all processing tasks are *idle* -or *absent*. The Supervisor is responsible for checking the hidden states of the Distributor, Collector and processing tasks. The *empty* state of the Distributor and Collector is self-explanatory. The *idle* state for the processing task is reached after attempts to acquire another event from the Distributor have failed and a pre-determined timeout has been reached. Clearly, this timeout is only

1. by calibration, we mean both electronic and geometry information

activated in the context of a *stop* sequence. A second global EH timeout mechanism must be implemented so that the EH is reset if the *Configured* state is not reached after a given time period. The definition of actions to be performed for resetting the EH is an issue of the prototype.

pause: *Running* → *Paused*

resume: *Paused* → *Running*

In our current view, the usefulness of these commands is not clear. If *pause* only means to freeze the system in its present state, it should be sufficient to stop the SFO and the SFI. The EH will reach a blocked state when either the Distributor buffer is empty or the Collector buffer is full. The policy about keeping or discarding events inside the EH when the *pause* command has been issued should be discussed.

When the transition from the *OK* to the *Bad* state is executed (generally corresponding to an unsolicited change in the state of the FSM), an error message is sent to the Run Control, giving the initial state of the FSM and the error code.

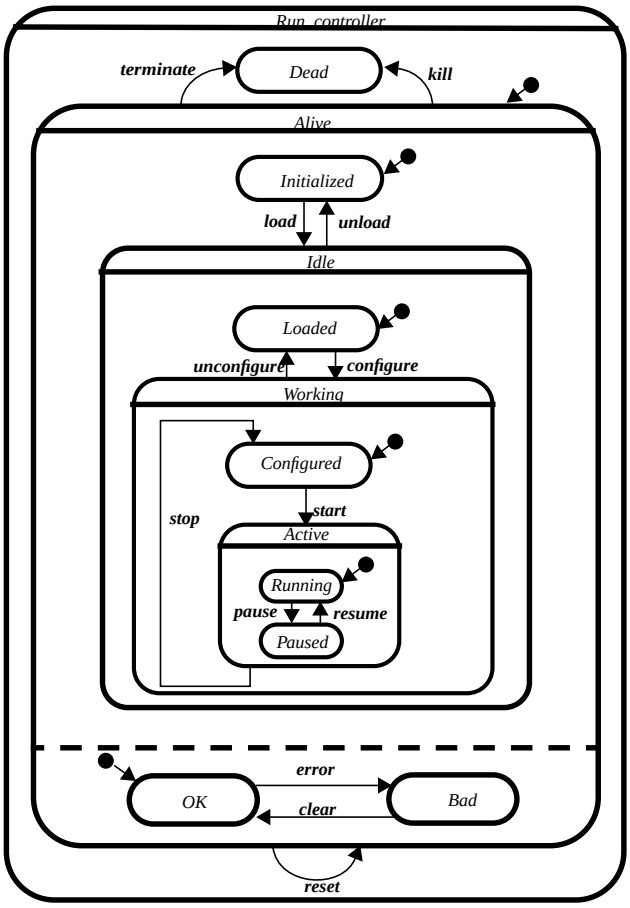


Figure 8 Hierarchical Finite State machine

2.5 References

1 The LDAQ in ATLAS DAQ prototype -1, ATLAS DAQ Prototype-1 Technical Note 44, http://atddoc.cern.ch/Atlas/Notes/044/Note044-1.html

2 A High Level Design of the Sub-Farm Event Handler, ATLAS DAQ Prototype-1 Technical Note 61, http://atddoc.cern.ch/Atlas/Notes/061/Note061-1.html

3 Users Guide and Implementation of the Message Reporting System for the Atlas DAQ Prototype -1, ATLAS DAQ Prototype-1 Technical Note 59, http://atddoc.cern.ch/Atlas/postscript/Note059.ps

4 Event Handler Supervisor High Level Design, ATLAS DAQ Prototype-1 Technical Note 92, http://atddoc.cern.ch/Atlas/Notes/092/Note092-1.html

Chapter 3
Software detailed design and generic implementation

3.1 Data flow through the Event Handler

We describe in this section the software implementation of the distributor, collector and processing task. The work presented here has been developed in the framework of the PC commodity prototype [5]. Care has been taken to separate the communication protocol from the application layer so that new protocols and hardware architectures can be easily tested.

3.1.1 User requirements

The experience of the first implementations has led us to define more stringent user requirements than the ones presented in Chapter 2. Since they are more related to the implementation, we have listed them here rather than in the previous chapter where the more generic requirements have been presented.

3.1.1.1 Event transfer between component and with SFI-SFO:

1. *Shall* allow to use any event transfer protocol.
2. *Should* support client-server connection resets, even in a middle of an event transfer
3. *Shall* support process relocation if the architecture is distributed.
4. *Should* provide automatic reconnection of clients.
5. *Shall* support binding at run time (dynamic) between components.
6. *Shall* ensure that events are recoverable in case of error: corrupted data, incomplete tranfer, processing task crash, etc.

3.1.1.2 Component control

1. *Shall* allow to use any component control protocol.
2. *Shall* support binding at run time (dynamic) with controlled components.
3. *Shall* support dynamic component parameters modifications.
4. *Shall* support stopping and restarting a component.
5. *Shall* support resetting of components.
6. *Shall* provide information on the component activity: throughput, number of events that traversed it, number of event contained in the component, number of events rejected by the processing task.
7. *Shall* inform a supervisor of errors or key steps in the component activity.

3.1.1.3 Component behaviour

1. *Shall* be independent of the event transfer and control protocol used.
2. *Shall* work with client or server for input or output.
3. *Shall* share the same control interface whatever its behavior is, including activity information.

3.1.1.4 The Event Filter

1. The distributor *shall* direct events toward type specific processing tasks
2. A processing task *shall* be associated to a single event
3. *Shall* support dynamic reconfiguration
4. Failure of a processing task *shall* not have any repercussions on the system functionality

3.1.2 Component design

All components can be seen as simple FIFO queues of a given depth.

Input and output behaviour of components may be client or server. A client can connect to only one server, send requests and wait for an answer. A Server can handle multiple client connections and wait for requests that it will process. Many different event transfer protocols and component control protocol may be used, while the behavior of the component remains unchanged.

This leads us to split components in three different type of **elements**. The **Core Element** in which we implement the component specific behavior; **IO Elements** in which we implement the protocol specific part of event transfer and a **Control Element** in which we implement the protocol specific part of the interaction with the supervisor.

IO Elements will be declined into input and output elements and each one of them into client and server elements. A Core Element must differentiate a client with a server. It must make a difference between input and output though.

A component will thus be an aggregation built at run time of these different elements.

In order to use a new event transfer protocol we only need to implement the corresponding IO Element classes.

We will have as many Core Element as we have different types of component behaviour and as many Control Elements and IO Elements as we have protocols to use or test.

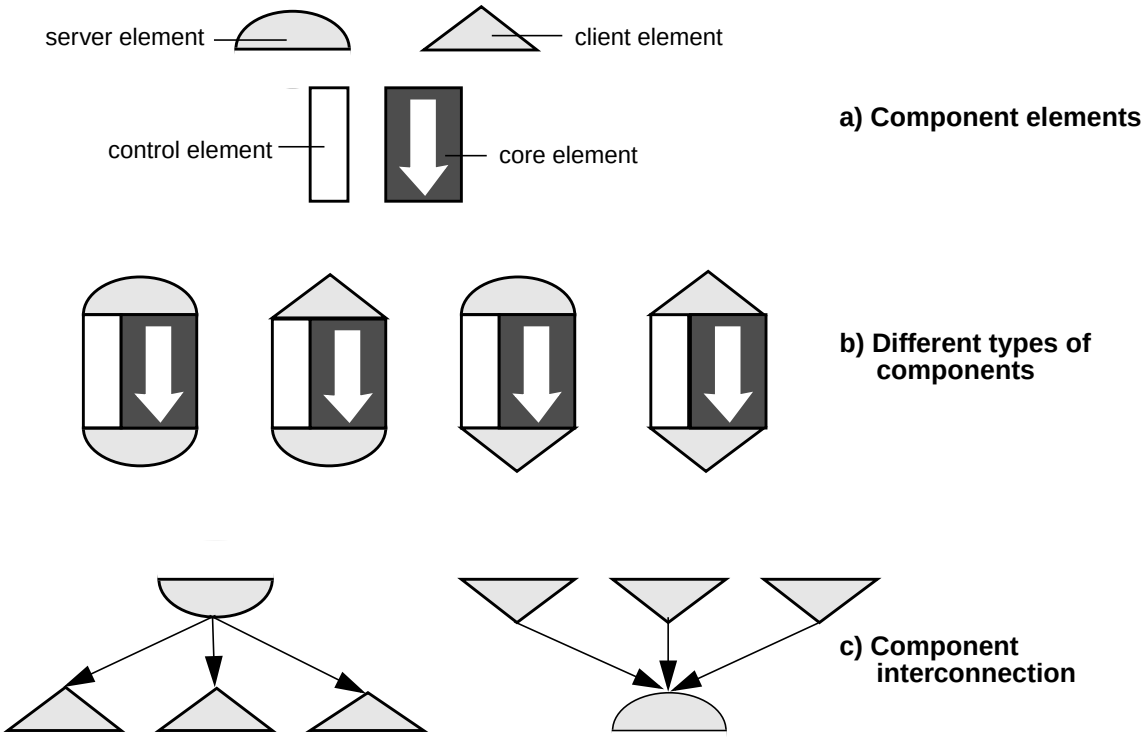


Figure 9 Component design

Figure 9 shows the design of the components, how the different elements can be aggregated into components and how components can be interconnected. Figure 10 shows a detailed view of a distributed implementation of the Event Filter. Data is pushed by the SFI towards the server input element of D1. A thread gets events from D1 (a server output element) and sends them to the proper D2 server input element. Events are then extracted from D2 by the client input element of D3. Processing tasks extract events from the server output element of D3 and pushes the selected ones to the server input element of C1 which itself pushes events towards C2 from which they are pulled by the SFO.

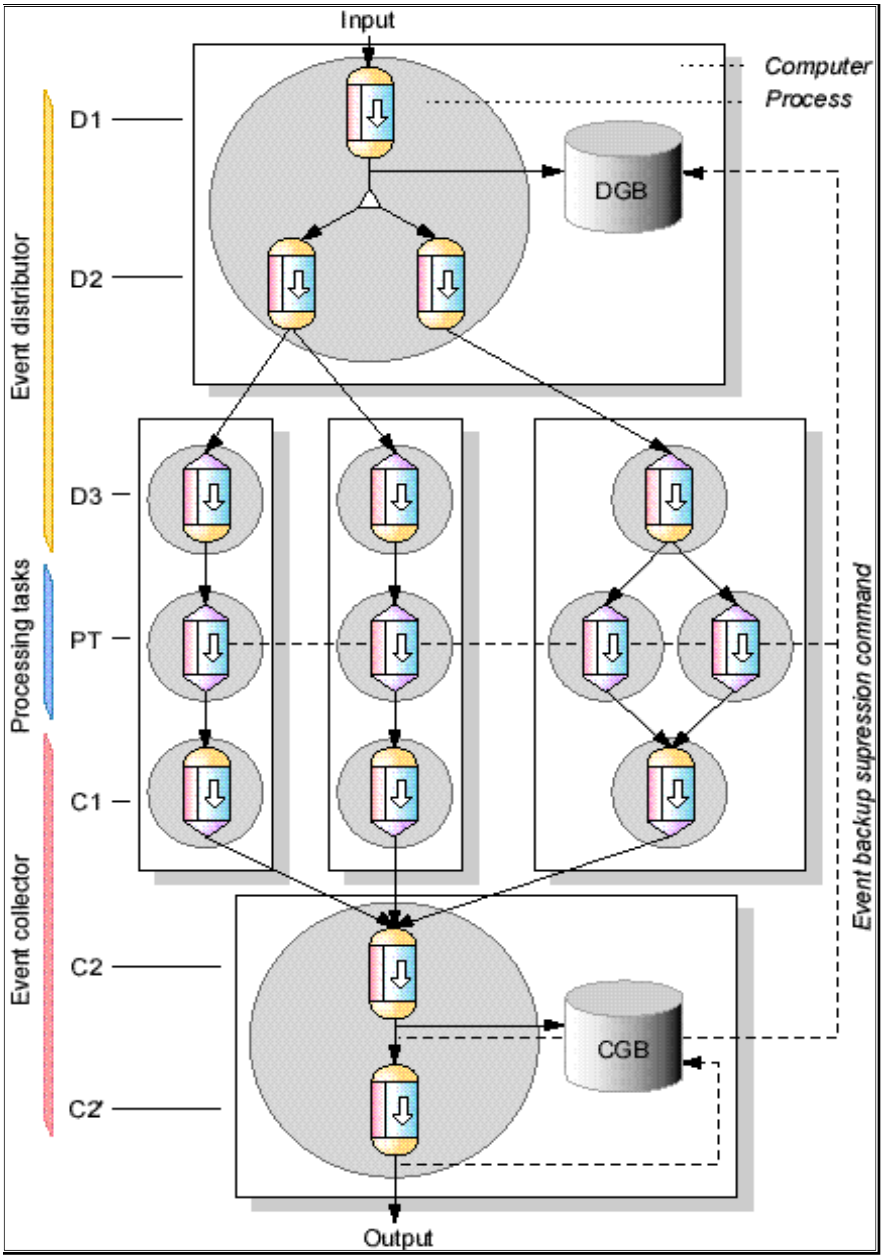


Figure 10 Example of a component implementation to realise a sub-farm

The different clients have the knowledge of the location of the server to which they are connected via a dedicated *Naming Service*. Communication between the components and the naming server is done using the UDP protocol.

3.1.3 Element states and methods

3.1.3.1 States

When elements are built, they are in the NULL state. They are subsequently assembled into an aggregated component, and are then in the INIT state. In this state we can initialize the component by calling the `init` method of the CoreElement. After this call, the component enters one of the following states : NOT_READY, READY or BROKEN.

The BROKEN states means that a fatal error occurred while initializing the component from which it cannot recover. The component should be destroyed.

The READY state means the component is operational and running. The NOT_READY state means that the component is not operational and not running. This happens in the case of a recoverable error, for instance when a client IOElement tries to connect to its server. At any time the Element may switch back and forth form READY to NOT_READY.

While READY or NOT_READY, the component can be stopped and restarted. In the later case, it returns to its previous state.

The status of each element can be expressed in one byte. It is a combination of 3 flags: InitFlg, NotReadyFlg, StoppedFlg. The element states are therefore expressed as: NULL = (0,0,0), INIT = (0,0,1), READY = (1,0,0), NOT_READY(1,1,0), NOT_READY_STOPPED = (1,1,1), READY_STOPPED = (1,0,1), BROKEN = (0,1,x).

3.1.3.2 Element methods

Details of the methods can be found in [5]. We only list here the name and the functionality of each public and private methods. Let us note that only the CoreElement has public methods which are the callable methods of the component.

3.1.3.2.1 Core Element

Table 4 Core element public methods

`get () : event`	← event input
`put (event)`	← event output
`init () : bool`	← control
`start () : bool`	
`stop () : bool`	
`reset () : bool`	
`setParam (params)`	

Table 4 Core element public methods

`getParam () : params`	
`getStats () : stats`	← monitoring
`getDeltas () : deltas`	
`status () : int`	

The Core element is basically a FIFO queue with possibly additional functionalities (eg processing task). The private methods allow to manage space in the queue. They are schematized in Figure 11 and summarized in Table 5.

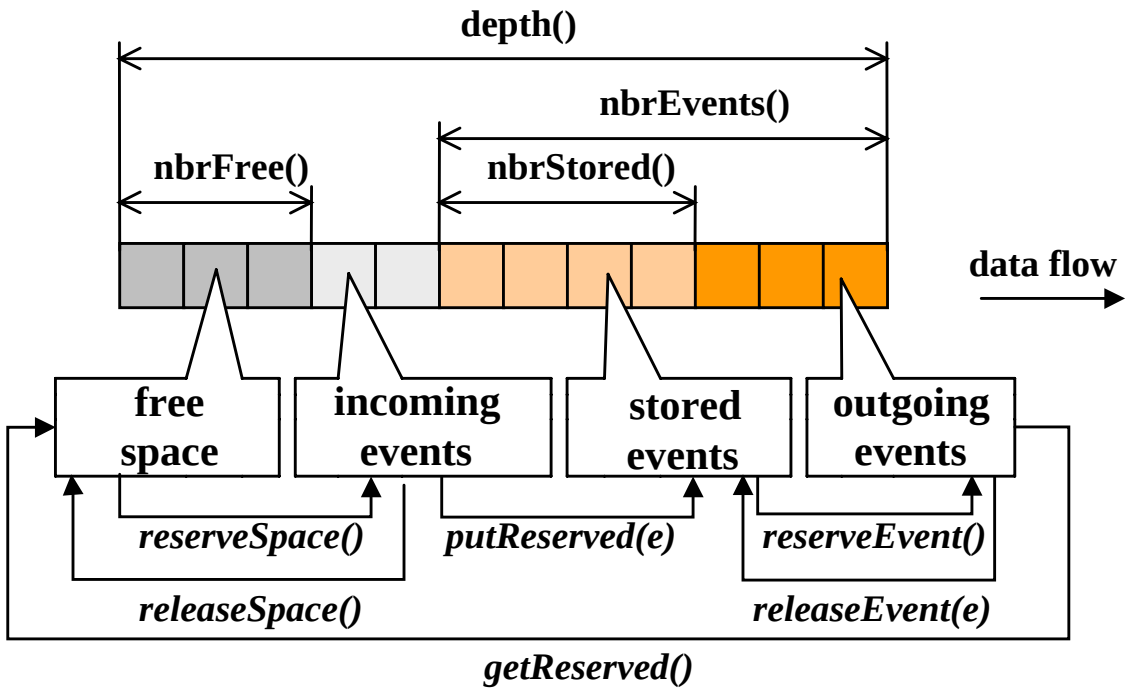


Figure 11 Low level operations on the CoreElement FIFO queue

The queue is divided in 4 logical zones. The methods quoted in the lower part of Figure 11 are used to move events from one division to the next one. Let us note that all operations are reversible. *Incoming* events are events for which space has been reserved (in the case of a server) and which are not yet fully copied in the queue. Similarly, *outgoing* events have been requested for output and have not yet been fully copied to the client. The methods in the upper part of the figure allow to know the actual size of the dynamic zones.

Table 5 Core element private methods

`report (`*`message`*`) : `*`bool`*
`reserveSpace () : `*`bool`*
`releaseSpace ()`
`putReserved (`*`event`*`) : `*`bool`*
`reserveEvent () : `*`event`*
`releaseEvent (`*`event`*`) : `*`bool`*
`getReserved () : `*`bool`*
`eventAvail ()`
`spaceAvail ()`

The `report` method will simply forward the message to the CtrlElement. The `reserveSpace` and `reserveEvent` methods are blocking functions. They will return if the request has been satisfied or an error occurred

3.1.3.2.2 IO Elements

Table 6 IOElement private methods

`stop ()`	← all IO elements
`start ()`	
`reset ()`	
`status ()`	
`setTarget (target)`	← client only
`spaceAvail ()`	← input only
`eventAvail ()`	← output only

The method `setTarget` allows to change the server from which a client may get or put events. A target name is a string. The CoreElement should keep track of the current target of an IOElement. `Status` will specify if a client is connected with the ReadyFlg set if the protocol is connection oriented.

3.1.3.2.3 Control Element

Table 7 CtrlElement private methods

`stop ()`
`start ()`
`reset ()`
`status ()`
`setTarget (`*`target`*`)`
`report (`*`message`*`)`

`Report` will send the message to the specified target. The target can be changed by `setTarget`. The Core Element is supposed to keep track of the current target.

3.1.4 Component implementations

The so-called "Version 3" of the code, which is publicly available, has been written in C++. Care has been taken to clearly separate the functionality layer from the communication layer. Code specific to every component has been kept in a separate directory (`d1d2`, `fifo`, `pt` for the specific components, `ptcalorec` for the benchmarking code running in the processing tasks, `nserver` for the naming service). All components have been built from elements contained in the `libcomponent` library. Additional libraries are available, which contain the transport specific elements: for the PC prototype (Chapter 4), we have developed two libraries, one based on TCP, the other on the CORBA compliant ILU package. Both are derived from a model package. Figure 12 shows the Package Diagram of the Data Flow.

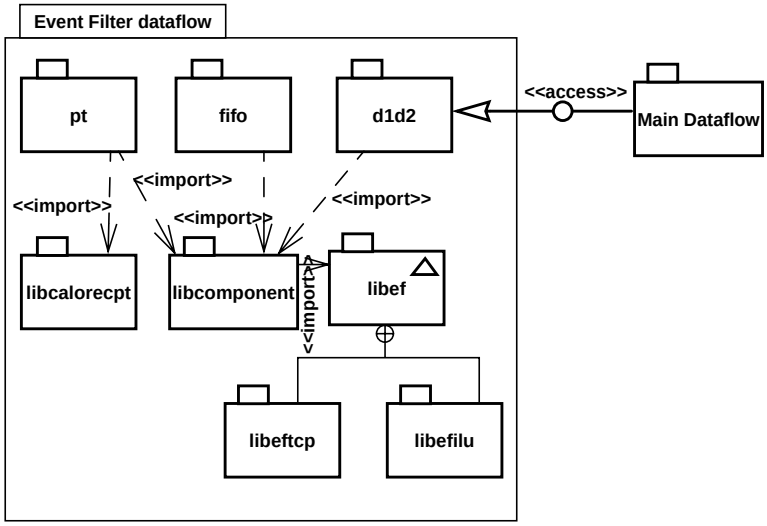


Figure 12 Package Diagram of the Data Flow

3.2 Event Handler API

3.2.1 Introduction

This section is a summary of Technical Note 98 [6]. It describes the proposed application programming interface between the DataFlow and the Event Filter. This API provides the dataflow with an interface which is independent of the sub-farm implementation. The efficiency of the approach has been demonstrated when the three prototypes described in the following chapters have been integrated with the main data flow.

For the purpose of overall design flexibility, two options are proposed for the event sending mechanism from the SFI to the Distributor and for the event sending mechanism from the Collector to the SFO. Thew so-called *Blocking* and *Non-Blocking* schemes are defined as:

- Blocking: the action of sending a message prevents the sending party from executing any other action until the message has been (successfuly or otherwise) sent
- Non-blocking: after initiating the dispatch of a message, the sending party may execute any other unconnected action, returning later to check the status of the dispatched message

Message exchanges are executed in the hypothesis of *synchronous* operation: a message may be sent by a party only if the other is prepared to receive it and the receiving "process" has been started.

3.2.2 SFI - Distributor functions

The proposed functions are:

- **`EH_OpenDist`**: open connections between the SFI and the Distributor - initialisation
- **`EH_CloseDist`**: close connections between the SFI and the Distributor
- **`EH_ResetDist`**: reset SFI-Distributor communication links
- **`EH_BlkReady`**: check that the Distributor is ready to receive an event (blocking call)
- **`EH_BlkSend`**: send an event from the SFI to the Distributor (blocking call)
- **`EH_NblkReady`**: check that the Distributor is ready to receive an event (non-blocking call)
- **`EH_NblkSend`**: send an event from the SFI to the Distributor (non-blocking call)
- **`EH_StatusDist`**: check transmission status of a non-blocking send

3.2.3 Collector - SFO functions

The proposed functions are:

- **`EH_OpenColl`**: open connections between the SFO and the Collector - initialisation
- **`EH_CloseColl`**: close connections between the SFO and the Collector

- **`EH_ResetColl`**: reset SFO-Collector communication links
- **`EH_BlkPending`**: check that the Collector is ready to receive an event and return the number of pending events (blocking call)
- **`EH_BlkRecv`**: receive an event from the Collector to the SFO (blocking call)
- **`EH_NblkPending`**: check that the Collector is ready to receive an event and return the number of pending events (non-blocking call)
- **`EH_NblkRecv`**: receive an event from the Collector to the SFO (non-blocking call)
- **`EH_StatusColl`**: check transmission status of a non-blocking receive

3.2.4 Example of timing chart

The chart illustrates the overall function call flow as seen from the Dataflow point of view. It does note take into account the internal operation details of the Distributor and Collector. In the present implementation, the readiness of the Distributor and Collector are checked by non blocking calls while the transfer operations are performed by a blocking call.

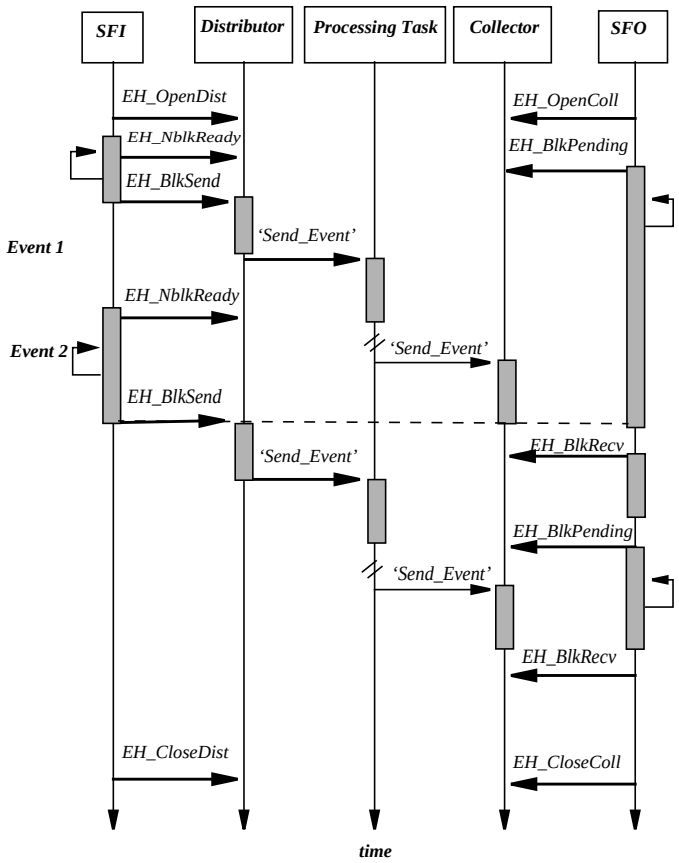


Figure 13 Sequence Diagram of the interaction of the Event Filter with the Main Data Flow

3.3 Supervision

The supervision of the sub-farm as well as the one of the whole Event Filter is much more related to the adopted technologies for the prototype and the supervision than it is for the data flow. Therefore no generic implementation of the Supervision has been undertaken up to now. The different implementations used by the different prototypes will be presented in the corresponding chapters.

3.4 References

5 Event Filter Farm Component Implementation Design, ATLAS DAQ Prototype-1 Technical Note 132, http://atddoc.cern.ch/Atlas/Notes/132/Note132-1.html

6 The Event Handler API, ATLAS DAQ Prototype-1 Technical Note 98, http://atddoc.cern.ch/Atlas/Notes/098/Note098-1.html

Chapter 4
Commodity PC prototype

4.1 Introduction

Commodity component computers offer an ever increasing performance to price ratio. Communication is not expected to be a bottleneck in an Event Filter sub-farm since applications running will be CPU limited. The distributed nature of the Event Filter complies rather well with the hardware architecture of farms of PCs. Scalability and hardware upgrades should be easy to perform. However, management of very large configurations will be very complex and will necessitate local in-house developments (provided by constructors in the case of SMP-like commercial machines). Hardware reliability is likely to be weaker than for a fully commercial solution.

In order to help to understand some of the problems quoted above, a PC prototype study has been launched. It uses bi-processor Pentium machines running Windows NT. Presently, Barcelona, the CERN IT division and Marseille are involved. The adopted strategy consists of developing an implementation of the adopted design on a small scale prototype and performing then the scalability tests at CERN on dedicated large configurations.

4.2 The available testbeds

4.2.1 The Marseille farm

This prototype is dedicated to software development and debugging on a small scale configuration (4 bi-processor machines). CPUs are Pentium II running at 450 MHz. Each machine has 64 Mbyte of dynamic memory. They are connected together via a 100 Mbit/s Ethernet switch (3Com 9300). A SUN workstation (SPARC Ultra-1 running Solaris 5.7) is connected to the farm by the same switch and is used to inject events into the farm.

4.2.2 PCSF

The PCSF cluster [7] composes the NT / CORE structure in the CERN Computing Centre. This cluster comprises one master node (the master computer PCSF901 is a Windows NT Server), one install server running Windows NT 4.0 server (called PcServer1) and 35 nodes running Windows NT 4.0 Workstation.

All machines have Tyan Titan ATpro motherboards with Dual Pentium Pro 200Mhz with 64MBytes RAM (rated at 8 SPECint95), or Intel DK440LX motherboards with Dual Pentium II 300MHz with 128MBytes RAM (rated at 11.6 SPECint95), Adaptec AHA-2940 or AIC-7895, two 4GB SCSI disks, low cost PCI video card (S3 Diamond), and an Intel Pro 100B ethernet PCI board with boot PROM.

All PCs use a multiplexer allowing to have only one video, keyboard and mouse for the whole cluster.

All PCs are attached to two Fast-Ethernet 3-COM switches (daisy-chained) with an upper Gigabit link. This provides a very good connectivity to the CERN CORE network as well as a high-bandwidth for internal transfers in the PCSF cluster.

The masters can be controlled remotely via Remote Control Tools (e.g. PCanywhere, PCDuo).

All systems are monitored by the CERN exception monitoring system, called CNSURE. The system generates alarms if one node is down or has lost some of its functionality. These alarms are displayed in the operator room and immediately taken into account to reduce as much as possible down-time.

All nodes are configured identically:

- same version of O.S. e.g. Windows NT4 Workstation and S4
- same monitoring mechanism
- same software installed locally (LSF for batch scheduling, Perl, SHIFT Software, etc...)

Software can be distributed under the Administrator account across all nodes from a single node using home-made scripts.

Configurations including 15 machines (Pentium II 300 MHz with 128 Mbyte RAM) have been made available to ATLAS during 4 periods of one week. We also had access to a SUN Enterprise 450 workstation connected to the farm by the Gigabit Ethernet link. This workstation was used to inject events into the farm.

4.2.3 EFF

From November 1999, we have used a farm entirely dedicated to Event Filter tests for the LHC experiments. This farm has a configuration similar to PCSF. The CPUs are Pentium III 550 MHz with 128 Mbyte RAM (Siemens CELSIUS 630 motherboard, rated at 24.4 SPECint95). 10 machines were available during our tests. Following a recent upgrade, the permanent configuration comprises 25 machines. Machines from other farms would enable us to have temporary configurations of about a hundred of machines.

Two additional machines equipped with a Gigabit Ethernet interface have been used for injection of events: a bi-processor PC (Pentium II 450 MHZ, motherboard Intel 440BX, 128 MB RAM) running Linux, and a SUN SPARC Ultra 4 (quadri processor SPARC II 450 MHz, 512 MB RAM) running Solaris 2.6.

4.3 The data flow implementation

The data flow implementation for the PC prototype follows exactly the design described in Chapter 2 and the generic implementation of Chapter 3. It has been written in C++. The architecture of the farm (number of active nodes, number of processing tasks per node, etc.) is given in a configuration file and the internal physical characteristics of the components (FIFO depths, executable binary files, etc.) are passed at run time as parameters of the launching command contained in the configuration file. Some parameters may be changed dynamically while the farm is running such as the run number.

4.3.1 ILU protocol

CORBA (Common Object Request Broker Architecture) [8] allows to invoke remote object methods in the same way as local object methods making its usage transparent for the programmer. In C++ it is implemented by use of a local object (Proxy Object) representing the remote object. Those Proxy Objects have the same methods as the remote object they represent but their method implementation handles the transfer of parameters and results with the remote object. On the remote computer, a server, named an *Object Request Broker* (ORB), will dispatch invocations to the appropriate object instance, the remote object.

Remotely accessible objects, sometimes named *Server Objects*, build up a special object reference allowing to locate the object in the Internet. This reference is called the *Internet Object Reference* (IOR). A user may then create a Proxy Object to access the corresponding remote object by use of this IOR without needing to know where the object is effectively located. This is why CORBA is sometimes referred to as an object communication Bus.

Communication between Proxy Object and server object is done by use of the IIOP protocol. It is thus possible to implement Proxy Object in any programming language provided it respects the IIOP protocol.

The simplicity of communication with remote objects by making it equivalent to a method call has a drawback. On the Server Object side, results are normally sent back to the Proxy Object when returning from the method. However, this may fail in real life on some occasions, but the Sever Object has no way to know this, even if the transport protocol detects it, because it is the ORB which will handle returning the result.

So in our context, if a Server Object is to return an event on an user request, the Server has no way to know if the client received the event or if it was lost. The question therefore arises: may the server delete or not the local copy of the event? Of course this problem could be easily circumvented, e.g. by adding acknowledgement methods. But this increases message exchange rates, intermediate states in the data exchange and ends up in redundancy in communication reliability information. This means that even if we use a reliable low level

communication channel, as for instance TCP, we must consider, from the Server Object point of view, the return of results in the CORBA context as insecure because we are not able to catch a failure although the ORB will have this information. There are some CORBA implementations which propose to solve this problem but they are beyond the current standard and are implementation dependent.

ILU [9] is one of the many implementation of CORBA. It is developed at Xerox's Palo Alto Research Centre and made available as freeware. The latest ILU release we used for our test was ILU 2.0 alpha 14 on Windows NT. ILU does not provide any means for a Server Object to catch a return failure.

In our communication layer using ILU we used multithreading and blocking calls on Windows NT. In such type of usage of ILU, we encountered some problems with this latest release. But this should not interfere with the decision to use CORBA or not for the data flow.

We also tried to use CORBA to exchange data between component implemented in C++ and the supervisor implemented in Java. But we had many problems of compatibility with IIOP implementation and interpretation. We then subsequently used standard and simple ASCII over UDP message exchange to communicate between the component and the supervisor.

4.3.2 TCP protocol

We have implemented a communication layer of the Data Flow programme to use native and standard TCP. Events are transferred as an opaque byte sequence through a TCP connection.

A simple implementation would rely on the TCP back pressure mechanism to manage data flow control. We did not test this strategy because we feared that this may lead to saturation of internal system buffers and interfere with traffic between non saturated components. We thus decided to support "on demand" event transfer where events are only sent if a request is received. This is a two way message exchange, the second being the event itself. We call this pulling an event in opposition to pushing event that occurs when a client wants to send an event to a server object. In that case the client send a request for space to which the server responds with an event request as soon as it has space to receive the event. This is thus a three way message exchange and is less efficient than the pulling transaction. However pushing event occurs only in the collector and will thus support a much smaller network traffic than the distributor where all the pulling transactions occur.

4.4 The supervision implementation

When implementing the Supervisor of the PC prototype, our aim was to provide a supervision system independent of the platform and of the operating system, scalable and able to adapt to different architectures and implementations. We have chosen to use the JAVA Mobile Agents technology which seemed able to fulfil these requirements. In parallel to these developments, a "mini" supervisor has been written in C++. It allows to visualize in a very simple way the state of the components and some statistics, and has been used in the debugging phase.

4.4.1 JAVA Mobile Agents

The Java Virtual Machine and Java's class loading model, coupled with several of the Java features, of which; serialization, remote method invocation, multithreading, and reflection are the most pertinent, have made building mobile agent systems a fairly simple task [10].

Java Mobile Agent systems have a number of key characteristics:

- all Java Mobile Agent systems provide an agent server, which is a contact point on a given machine (Figure 14). Those server objects act as warehouses, or workplaces into which agents move and in which agents act. A server provides a means of hosting and managing its own agent in an environment that is secure from malicious agents.
- agents can migrate from server to server, carrying their state with them. After having moved into a server, an agent becomes a local user, and it can do everything that a local user can do, such as getting system resource information (*e.g.* process, disk, memory, cpu, network usages), create / delete processes, etc.
- agents can load their code from a variety of sources. In general, since all the agent systems use a specialized version of the Java classloader, they can load Java class files from the local file system, the Web, and ftp servers.
- they are 100% pure Java. This means that they should run on any computer with a compatible Java runtime library.

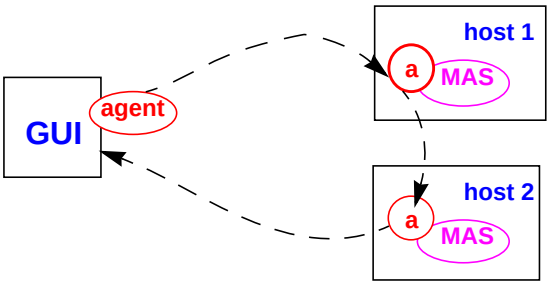


Figure 14 Use of Mobile Agent Servers (MAS) by JAVA Mobile Agents

ObjectSpace Voyager Core Technology [11] is an advanced, 100% Java Object Request Broker (ORB), based on the Java language object model.

Among others, some reasons why we decided to evaluate this framework are:

- three things in one: Voyager supports three types of communication: point-to-point, client / server, agent-based
- ease of use
- good performance
- very good scalability
- compactness (Voyager core classes are only 600 kB)

- many functionalities correspond to the needs of DAQ: space scalable group communication, publish/subscribe, event & listener, object persistence, integration with CORBA.

The implementation scheme is shown in Figure 15. There are 3 levels of monitoring: system supervision (at the machine level, possibly provided by the operating system), Mobile Agent for event data flow control and monitoring, offline analyse tools for monitoring data archived in a database.

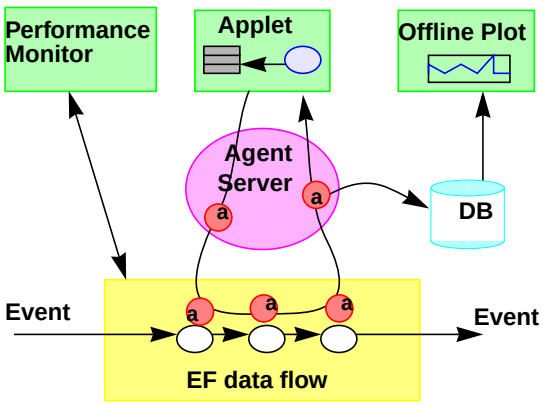


Figure 15 : Implementation scheme

Mobile agents collect information from the various supervised components. The information can be directly exploited by the user interface or can be stored in a database for subsequent timewise analysis.

Figure 16 illustrates how commands and status requests can be sent from the Control Interface to remote nodes by mobile agents and how status data is collected and returned to any monitoring interface. We have developed a unique interface for control and monitoring which can be used either in standalone mode or over the Web. Several copies of the interface run at the same time, but only one is allowed to perform control tasks. The agent server notifies all running interfaces when the EF status changes.

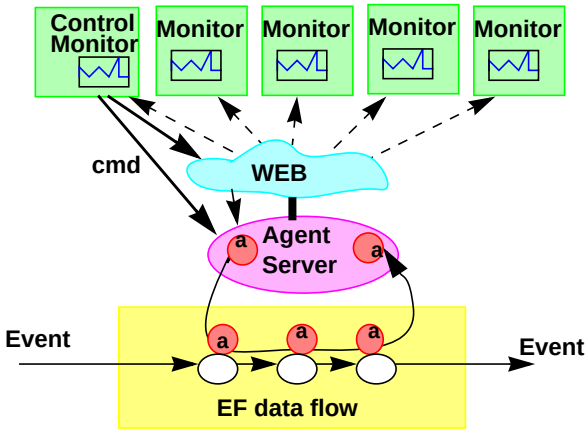


Figure 16 : Control scheme

4.4.2 Functionalities of the Supervisor

The GUI (Figure 17) can run in application mode or applet mode. A password-based access control is used for monitoring/control switching, ensuring that a unique GUI for control is enabled. Different viewers allow to display the entire EF hardware configuration and the event data flow structure. Control of the use of the archiving database is provided: user can switch on/off DB, change the rate of collection. An embedded mini browser allows to select configuration file over the web. It is possible to add/delete process at runtime. User can see agent travelling status via agent itinerary window. Online histograms and plotting packages are provided.

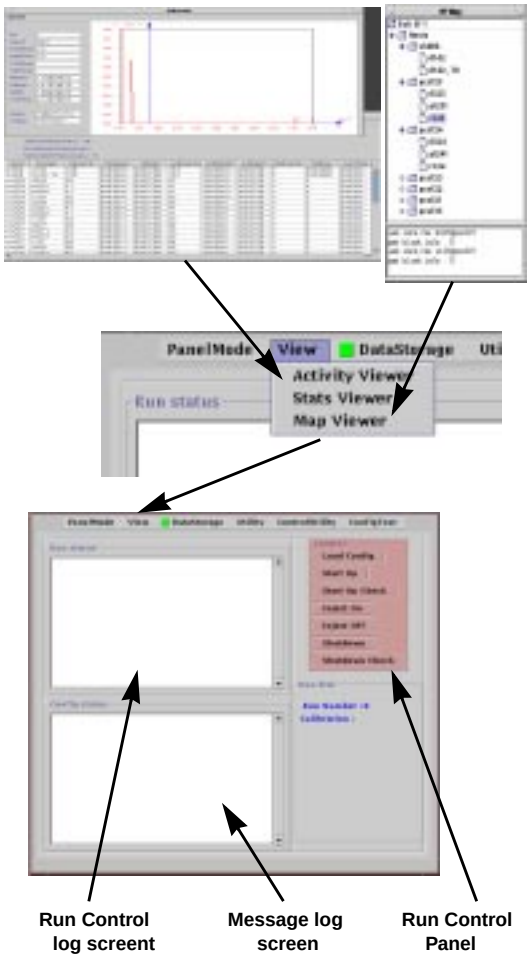


Figure 17 Graphical User Interface

4.4.3 Other possible technologies

We have investigated a new product called iBus//MessageBus from SoftWired AG [12].

iBus//MessageBus is a pure Java messaging middleware aimed at supporting applications such as content delivery systems, groupware, fault-tolerant client-server systems, and multimedia application.

iBus product implements the industry standard Java Message Service or JMS API. It is designed to plug into application servers and EJBs.

The basic communication paradigm of iBus is *publication/subscription* where messages are sent to named topics rather than specific processes or ports. Communication is typically one-to-many and asynchronous, although point-to-point communication and synchronous invocation are accommodated as well.

We intend to study this product in the EF environment for the following reasons:

- it has *no single point of failure*. It operates without special daemons or background services.
- alternatively, the *persistent message hub* provided in iBus/MessageServer can be used to support a central store of messages.
- a *request/reply* operation is provided, that work much like RMI which is two-way operation.
- it provides a *quality of service (QoS) framework* in which applications only pay for the services they need. Inside of one application (such as the supervision), the user can choose different QoS for different part of the application, for example debug message and alarm do not need same quality of service. The QoS includes reliable and unreliable multicast, reliable and unreliable point-to-point communication and failure detection. It is possible for user to add new qualities of service such as forward error correction or encryption.
- iBus provides a bridge allowing to access iBus from applications written in C, C++.

4.5 Performance

4.5.1 Conditions of the tests

The final throughput tests have been performed at EFF. Scalability and robustness tests have been performed both at PCSF and EFF. The processing task was performing the EM calorimeter reconstruction using the code specially developed for this purpose. Because the duration of the processing time was too short when compared to the expected one (~ some seconds), a more realistic one was simulated by running several times the same algorithm. This number could be fixed or set at a random value within a given interval. The rejection rate was set in the configuration file. Event transferred through the farm were 1 MByte in size.

4.5.2 Communication protocol

As a first remark, one can note that the server using the TCP protocol may detect improper event transfer, cancel the transaction and return it to the pool of available events. This was not possible with the CORBA implementation.

Second, the TCP implementation uses multiple connection state management in the server run by a single thread. In the CORBA implementation, blocking calls that would block on semaphores were used. The ORB needed to launch a thread per request. This blocking is also not interruptible when using POSIX semaphores. Besides this constraint, this leads to a much heavier system although very simple from the programmer's point of view. To circumvent these problems we may have implemented an "on demand" event transfer transaction on top of CORBA but the final system would be no longer simpler than the current TCP implementation. This would also add up message exchange between client and server and penalize throughput.

Since the number of operations to do on remote objects is very limited: `put(event)` and `get(event)` (and is not likely to change in the future), using CORBA to hide communication implementation brings no significant benefit.

This experiment and R&D work lead us to conclude that CORBA is not really needed and in some aspects not appropriate for event transfer in the event filter data flow. We need to combine performance with reliability and with a very small set of operations.

Use of CORBA would be more appropriate for control and supervision of the event filter farm.

4.5.3 Throughput

Events were injected into the farm via a SUN Enterprise workstation. Due to software limitations of the driver, the maximum transfer speed between the workstation and the farm was limited to 280 Mbit/s. Figure 18 shows the farm throughput as a function of the number of processing nodes. Each node was running 2 processing tasks and the throughput has been measured at the D1 level.

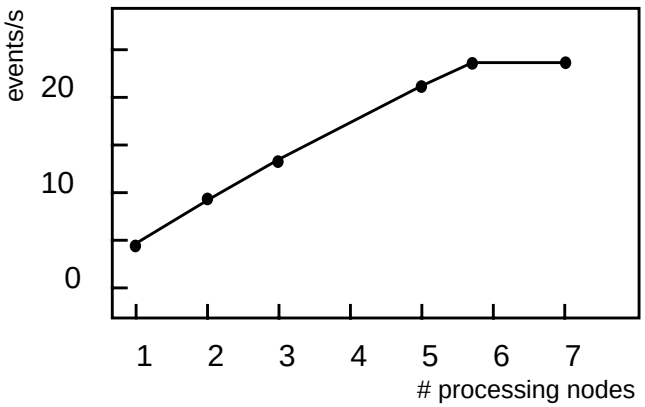


Figure 18 Farm throughput as a function of processing node (2 processing task are running on each node)

A maximum throughput of 24 events/s corresponding to approximately 200 Mbit/s is reached for 6 processing nodes. This throughput corresponds to 70% of the maximum bandwidth of the workstation interface to Ethernet. More studies should be performed to understand the limitations of the driver and of the farm.

4.5.4 Scalability

When many components are involved in a farm configuration, two bottlenecks can be foreseen: the naming service which must provide references for every client trying to connect to a server (in particular at starting time) and the supervision which has a lot of components to visit and to report on.

Configurations with increasing numbers of processing tasks have been launched on EFF. The naming service was running on a PC which had no other task than the distributor. Configurations with up to 56 processing tasks per node have been launched (525 components were then involved) without any problem. Voyager is able to send its mobile agents and retrieve all necessary information within less than 10 seconds. Startup of such a large configuration is performed in 60 s while shutdown and its associated checks is done in some 130 s. More conventional configurations including 10 PC and 2 processing tasks per node are started up in 5 s, shut down in 14 s and an information collection over the whole farm takes 0.5 s.

4.5.5 Robustness

Robustness has been tested against process crashes. The ability to restart a component on the fly, possibly on an other physical location has been verified in hundreds of occasions, faking a crash by killing the process. Restart of the process has always provoked the resuming of normal activity of the farm.

The ability of the DGB to ensure the security of data during the lifetime of the events inside the farm has also been tested. The implementation of the DGB was rather crude: a simple backup of the event on a disk (two stripe Seagate SD34572WD). During our tests several tens of thousands of events have been successfully processed. None has been found in the DGB at the end of the run. When a process crash was provoked, we have always found the corresponding event in the DGB at the end of the run. The price to pay for this security was the loss of a factor 2 in the throughput of the farm. Considering the strong dependency on the choice of the hardware implementation of the DGB, no attempt has been made to improve the performance for the time being.

4.6 Integration with the DAQ/EF-1 prototype

Full integration of the PC prototype with the DAQ/EF-1 prototype has been realized. Events sent by the SFI running in Building 40 at CERN have been sent to the farm running either in Building 513 (PCSF) or in Marseille and successfully bounced back to the SFO again running in Building 40.

Because Windows NT is no longer supported by the DAQ/EF-1 project, no formal test has been made to prove the integration of the prototype with the Backend software, namely the LDAQ of the sub-farm (Run Control), the Message Reporting Service and the Information Service. The simplicity of the scheme shown in Figure 19 makes us rather confident of our ability to do this easily.

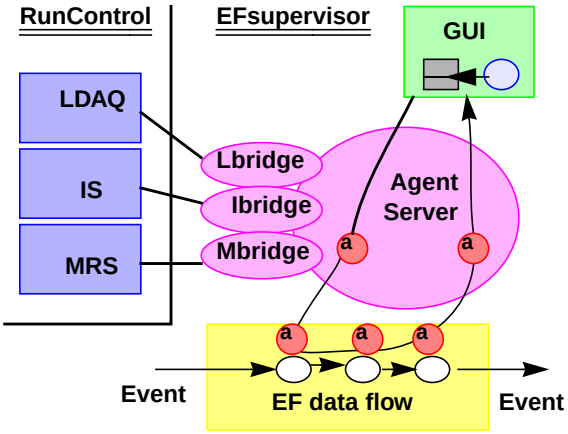


Figure 19 Bridge for the DAQ Run Control

4.7 Conclusions and outlook

The ability of the implementation of the data flow design in the distributed environment of a commodity component farm has been clearly established. The use of the JAVA mobile agents technology for the supervision has been proved to be very promising.

Further investigations should explore the use of the Linux operating system. More information on the hardware reliability is still needed. We also need some clarifications on the roadmap of the commodity component processors (the future of the Pentium family) as well as a more precise evaluation of the computing power required by the filtering activity in order to have a better idea of the final size of an Event Filter farm in such a configuration.

4.8 References

7 F. Hemmer et al., “PCSF - A PC based Simulation Facility running Windows NT” CHEP 98 Conference, Chicago, IL (USA), 31 August - 4 September 1998

8 Object Management Group, http://www.omg.org/

9 Inter Language Unification, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

10 Joseph Kiniry, Daniel Zimmerman - A hands-on look at Java mobile agents, IEEE Internet Computing, Vol.1, No. 4, July-august 1997, p. 23-30, http://computer.org/internet

11 Voyager - http://www.objectspace.com

12 Softwired Messaging Solutions http://www.softwired.ch/

Chapter 5
Symmetric Multi Processor prototype

5.1 Introduction

The Event Filter computing engine is organized functionally as a set of independent sub-farms, each connected to an output port of the EB switch, and its processing power is supplied by several processors working in parallel. In this context a complete sub-farm can be implemented on a single SMP machine [13]; the SMP architecture offers evident advantages in data sharing and transfer between the different hardware (and software) components: the main memory and many other system resources can be accessed symmetrically by all the processors through a very high speed system interconnect (system bus, crossbar switch, etc.).

5.2 Sub-farm implementation

In order to avoid as much as possible interferences of critical operating system aspects in the sub-farm code implementation itself (and obtain a better reliability of both hardware and software components) the prototype has been developed on a commercial SMP architecture with proprietary operating system, but the prototype has also been ported to an SMP commodity PC running LINUX OS. The technical choice has been a Hewlett-Packard SMP server running version 11.0 of the HP-UX operating system that provides kernel level POSIX thread and is POSIX 1003.1c compliant (draft 10). POSIX compliance allows for an easy porting of the code on other operating systems obeying the same standard. Indeed the code has been already ported in other environments (Solaris, Tru64-Unix, Linux). This sub-farm implementation has been tested on three different HP servers (4 CPU K220, 8 CPU N4000 and 20 CPU V2500) and on a 4 CPU PC Intel based (COMPAQ ProLiant 5500).

In order to achieve a better exploitation of the hardware resources, all the components of the sub-farm have been implemented within a single multi-thread process. Every sub-farm component is assigned a thread scheduled directly by the OS kernel (“1x1” scheduling model: to each user thread corresponds one thread in the kernel). One clear advantage is that, with this choice, load balancing, a critical parameter in the sub-farm operation, is automatically provided by the operating system scheduler.

5.2.1 Dataflow

The choice of the multi-threaded implementation stems from the fact that it eases in particular the communication and the synchronisation among the different components. The communication between the components can be achieved using the memory space of the process itself that is visible by all the threads: the Distributor and the Collector elements (objects D1, D2 and C2) are simply FIFO buffers containing the pointers to the events stored in the process memory space.

As a consequence, the event data-flow through the sub-farm is achieved by passing these pointers to the different components, whereas the real event does not change its physical location in memory. In this view, the IOElements of the components implement the passing of pointers and are executed sequentially by each thread that manages the component. The communication with the external Data-flow (SFI, SFO) is instead achieved implementing in the **IO Elements** of the component D1 and C2 the API described in Chapter 3.2 and based on CORBA ILU.

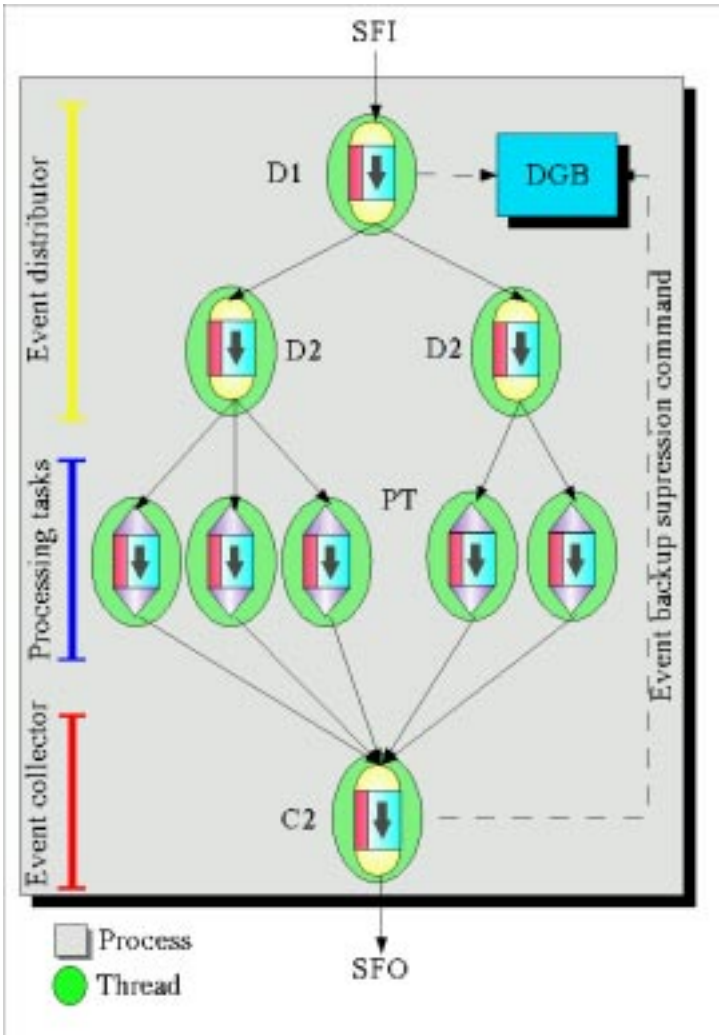


Figure 20 Subfarm design.

The functionalities required to the **Control Element** are automatically ensured by the POSIX thread library, that provides several system functions for the management of the thread associated to it, and by the fact that in this multi-thread implementation the component statistics are visible from the whole subfarm (global-variables). The addressing of each component from outside the subfarm is provided by an additional thread that manages the communications for all the subfarm components.

Figure 20 shows the subfarm design according to Version 3.

With reference to Figure 10 in Chapter 3, the D3 and C1 objects are redundant in this implementation: the processing task components get the events directly from the D2 buffer and send the accepted ones to the C2 component.

5.2.2 Supervision

The local supervisor provides the management functionality within the sub-farm. It controls all the sub-farm activities and provides a global co-ordination function for the sub-farm.

In the present configuration, the main thread controls a server socket and acts as a supervisor. Through a client socket it is possible to

- dynamically control the sub-farm operations as the initialization, starting, stopping, resetting and restarting of each sub-farm component in a totally co-ordinate fashion
- change some sub-farm parameters as the number of processing tasks, processing time, DGB activity, etc.

Moreover, it is also possible to monitor the sub-farm status. Since each component has a private set of counter variables that can be polled at any time, it is possible to monitor

- the state and the level of occupancy of the FIFOs
- the activity of each processing task (the number of processed, filtered, rejected events and the processing time)

5.2.3 Error handling

One of the must critical points of this design is the error handling. Since all the sub-farm is implemented in one single process, the failure of a processing task thread could lead to the crash of the entire sub-farm. The probability of this occurring can be drastically reduced by exploiting all the means available in the thread POSIX library. Indeed the synchronous signals, generated by the operating system as a consequence of a program error, are delivered directly to the offending thread. This thread can run itself the appropriate error recovering routine (signal handling routines and cancellation handling routines) and then kill itself preventing the loss of events.

In some cases, addressing errors in the common data memory space of the process may not lead to error signals: in this situation the error can induce faults in other threads at different times making a recovery strategy very hard to apply. This problem is currently under study and the key point is the evaluation of the probability of this event occurring.

5.3 Sub-farm performances

The general aim of the tests was to measure the global throughput and the scalability of the sub-farm and to check the software and hardware architectures.

The tests have been performed on all the available SMP machines varying the running conditions such as the number of processing tasks, the event size, the processing time (by looping many times the reconstruction task to simulate different realistic values), the number of processors and the use of the Distributor Global Buffer. The software executed by the processing tasks is a C++ version of the ATLAS Electromagnetic Calorimeter reconstruction software.

Every run consists of more than 20,000 events and in order to simulate the real ATLAS event size, their original size (~50 KB Monte Carlo data) is padded to 100 KB or to 1 MB according to the established type (“calibration” or “physics” respectively). Many of the results presented in the following for the K220 server are more extensively described in DAQ-1 Note 128 [14]. The same type of tests have been recently performed on the N and V servers to study in particular the scalability of the implementation.

All tests have been performed using a previous release of the prototype code, a non object oriented (written in C) version that, anyhow, is perfectly compliant to the high level design outlined in Chapter 2. The C++ version compliant to the common implementation design outlined in Chapter 3 is currently under development and will lead only to minor changes in the behaviour and in the performance of the prototype.

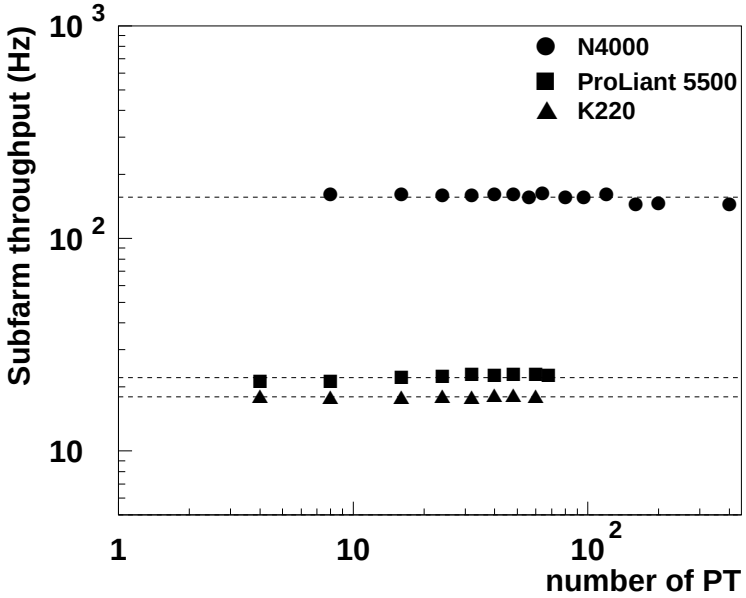


Figure 21 Throughput vs. number of PT on different machines.

5.3.1 Hardware

The prototype has been developed on a HP K220 server based on 4 120 MHz PA-7200 processors with 1 MB + 1 MB L2 cache and with 512 MB of RAM. The shared memory and external resources are concurrently available via the (64 bits @ 120 MHz) RunWay bus, with a sustainable rate of 750 MB/s.

The scalability test has been done on a 8 CPU HP N4000 and on a 20 CPU HP V2500 SMP servers powered by 440 MHz 64-bit PA-RISC PA-8500 processors with 1.5 MB L1 cache on chip (0.5 MB instruction + 1 MB data). The 8 CPU N4000 SMP server has an architecture based on two IA-64 system busses, with a total bandwidth of 3.8 GB/s, that connects the processors to the central memory controller which manages 8 GB of SDRAM; 240 MB/s IO channels ensure a total IO bandwidth of 5.8 GB/s. The architecture of the 20 CPU V2500 SMP server is based on a non-blocking 8x8 crossbar HyperPlane that provides 15.36 GB/s memory bandwidth with bi-directional 960 MB/s per port; the physical memory is 16 GB of SDRAM and the peak aggregate IO channel bandwidths is 1.9 GB/s [15]. We acknowledge CILEA [16], a computer centre located near Milan, for dedicating us the servers N4000 and V2500, allowing us to perform the necessary tests.

The intel-based platform is a COMPAQ ProLiant 5500 with 4 CPU PII Xeon clocked at 400 MHz and with 512 MB of RAM.

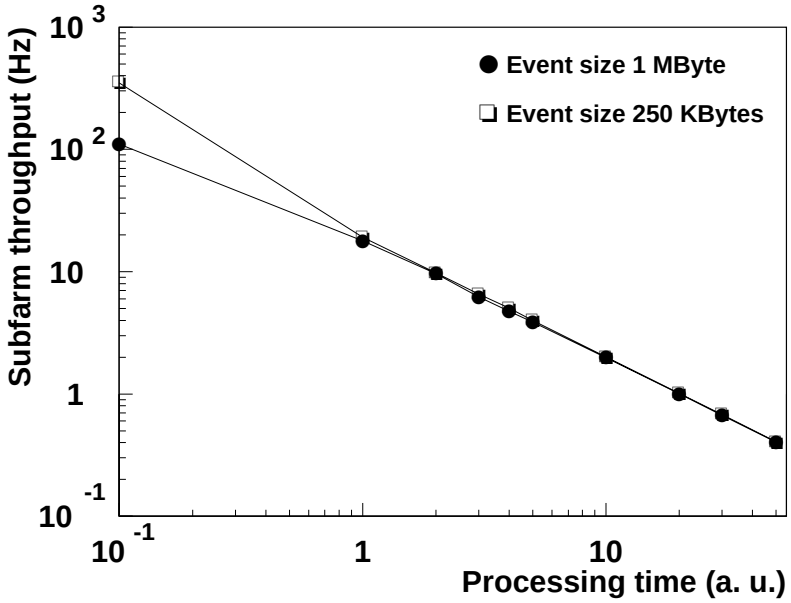


Figure 22 Global throughput as a function of processing time with different event size on K220.

5.3.2 Throughput

In order to prove the robustness of the software architecture a series of tests have been performed varying the number of “physics” processing task threads from 8 to 400 on the different available machines (HP K220, HP N4000 and COMPAQ ProLiant 5500). In this way the operating system has to balance a very large number of processing tasks: the results

obtained show that the global throughput is not affected by the number of running processing tasks. This is the proof that the software architecture is able to manage several processing tasks for each CPU node without any degradation in the global performance. In Figure 21 the throughput is displayed as a function of the number of processing tasks running on the sub-farm for the different platforms.

Then series of runs has also been taken modifying the processing time from 0 to 50 units: one processing time unit is ~220 msec on K220, ~40 msec on N4000 and ~100 msec on ProLiant 5500. The results show that, as expected, the throughput is inversely proportional to the processing time and independent of the event size for realistic processing time. On both the K220 and ProLiant 5500 this test has been performed with an event size equal either to 1 MB or to 250 KB (see figures 22 and 23), the results show that the throughput of both set of measures overlaps perfectly, except for the case at processing time close to 0. A processing time equal to 0 means that the events are not processed and are passed directly from the Distributor FIFO to the Collector one; in this case the throughput approaches 450 Hz on N4000, 100 Hz on K220 and 61 Hz on ProLiant at 1 MB (400 Hz on K220 and 100 Hz on ProLiant 5500 at 250 KB): these values measure essentially the copying time in memory. Indeed, approaching very small event sizes, the throughput reaches 60 KHz on N4000 and 7 KHz on K220 showing that the hardware architecture of the backplane interconnects does not limit the performances: these results show the absence of bandwidth limits and bottleneck on transfer.

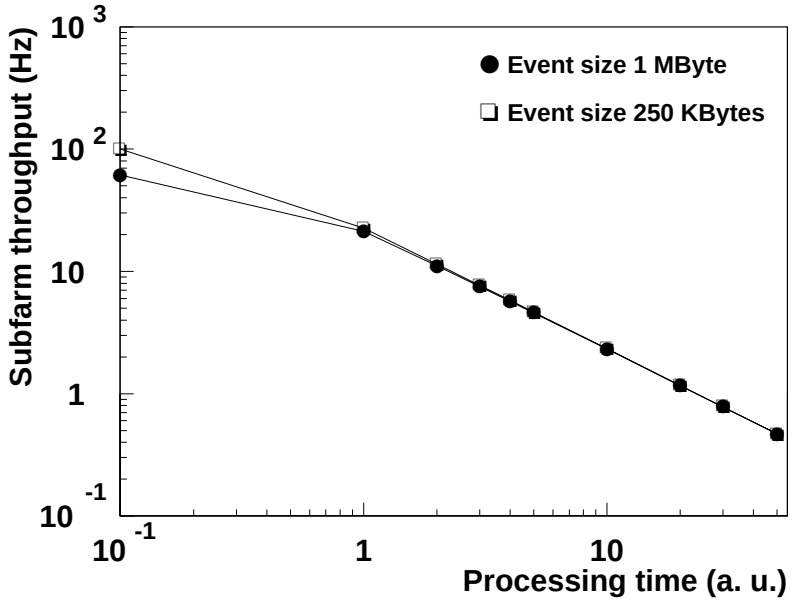


Figure 23 Global throughput as a function of processing time with different event size on ProLiant 5500.

5.3.3 Load balancing

The load balancing, which is a critical parameter of the sub-farm behaviour, is automatically ensured by the operating system scheduler and does not need any programming effort.

To better simulate the processing time expected with real reconstruction software, the sub-farm has been tested running the reconstruction task either with a fixed number or with a random number of loops (from 0.2 s to 4 s on a K220). Figure 24 show the results of the tests performed running concurrently two different types of processing tasks (“physics” and “calibration”) with an equal number of items (8 “physics” and 8 “calibration”). The achieved results prove that, with reasonable accuracy, each processing task is well balanced against the others of the same type and that the relative composition in the number of processing tasks does not affect the load balancing.

The load balancing is also ensured independently of the number of processing task as one can see in Figure 25: the operating system is able of balancing 400 threads at a level of few percent.

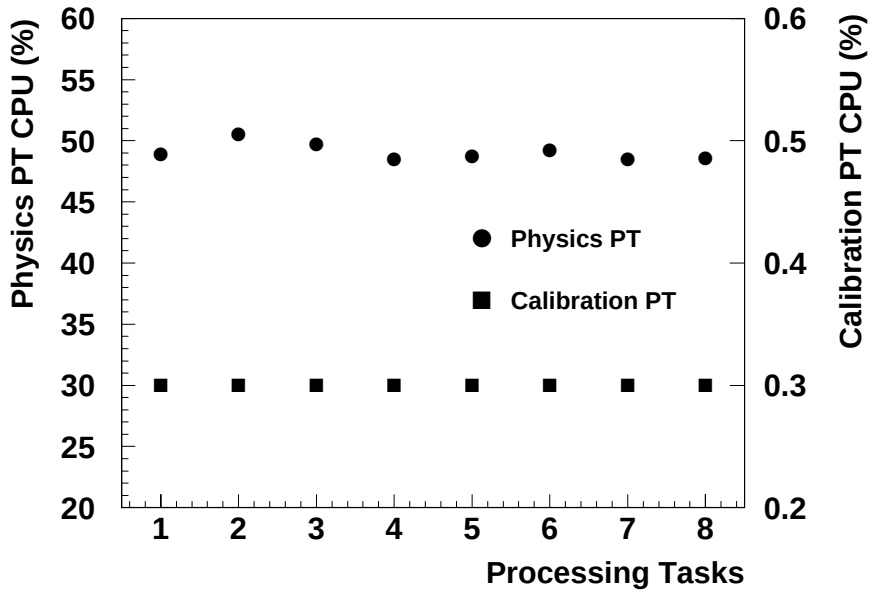


Figure 24 Load balancing of 8 “physics” and 8 “calibration” processing tasks on K220. Each processing tasks is well balanced against the others of the same type.

5.3.4 Scalability

One of the most interesting aspects of the multithread implementation is that, in principle, if there is no limitation in the hardware, the sub-farm multi-thread software allows for an exact scaling with the number of available processors in the SMP server. To check this important issue, a series of run increasing the number of the active CPUs in the V2500 server has been performed. It is indeed remarkable that the global throughput scales almost perfectly with the number of running processors, showing that an SMP sub-farm with up to 20 CPUs is feasible and that the V2500 hardware and the underlying operating system do not show bottlenecks of any kind. Different conditions as the event size and the processing time do not affect the results on scalability (Figure 26).

5.3.5 Robustness

As already noted in Chapter 3, the purpose of the Distributor Global Buffer (DGB) is to ensure that events are not lost during their passage through the sub-farm.

In the current implementation the DGB is a disk partition on which the events are stored as different files. They are stored as soon as they are received by the SFI and they are removed after being rejected by the processing tasks or disposed of by the SFO. In principle, another solution is to rely on the core-dump function of the operating system itself. This is particularly elegant because all the events in the distributor are in memory, and a copy of them will hence be found in the dump. This would avoid the time spent in disk writing for each event going through the sub-farm, but it has the obvious disadvantage of spending more time in the recovery process and of not addressing the crash of the sub-farm itself.

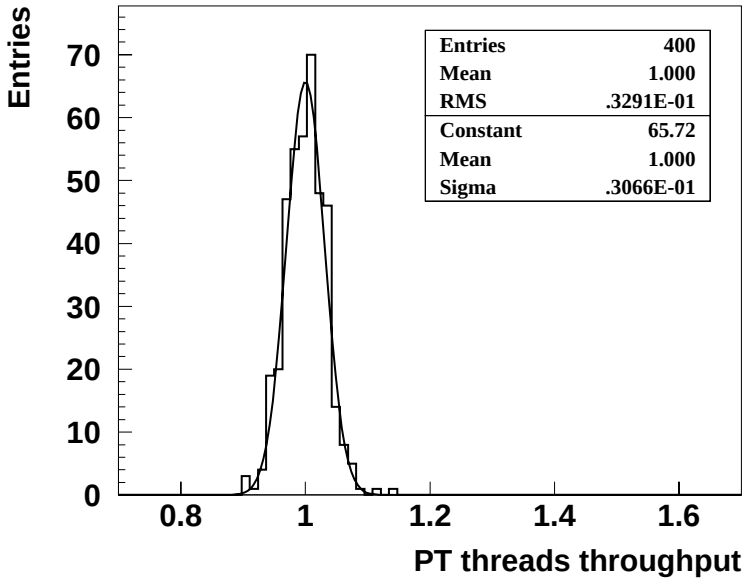


Figure 25 Distribution of the throughput of 400 processing task threads running on N4000. The load balancing is achieved at a level of 3%.

The correct DGB behaviour has been studied in many ways. First of all, at the end of run no event should remain in the DGB, and this has been found to be indeed the case. Moreover, in case of a simulated system crash, the events that were still to be processed have been found in the DGB and the recovery system embedded in the multi-thread implementation ensures that they are firstly processed when the sub-farm restarts before accepting new events from the Distributor: so no event is lost.

The use of the DGB influences the performance as expected: it reduces the global throughput at small processing time and becomes negligible increasing the processing time as it is shown in Figure 26.

Finally a long term reliability test has been performed: the sub-farm processed more than 4 millions events in 3 days without any problem.

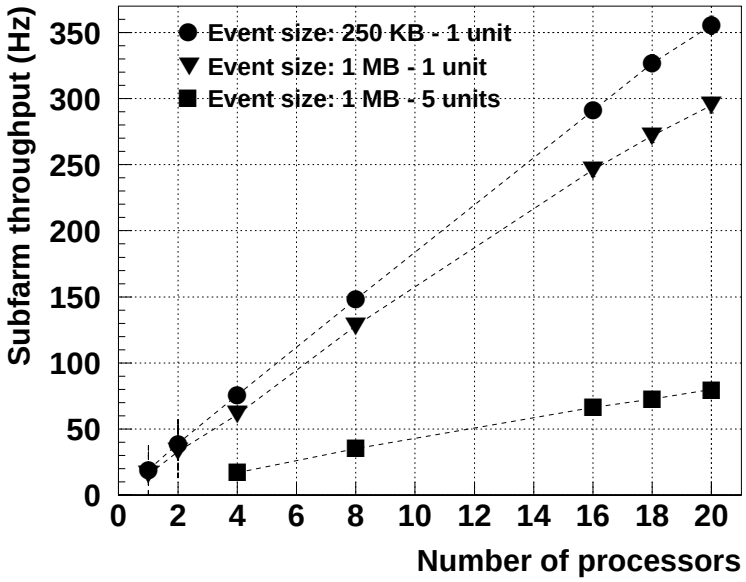


Figure 26 Throughput as a function of the number of CPUs with different conditions.

5.4 Integration with the DAQ/EF-1 prototype

The prototype has been integrated with the dataflow through the implementation of the Event Handler API outlined in Chapter 3.2; the transport protocol used is CORBA (ILU). The integration has been successfully tested running the sub-farm on a HP K220 server located in Pavia and the SFI and SFO elements on a RIO2 board located in the DAQ-1 laboratory in Building 40 at CERN: all the events sent to the sub-farm from the SFI had been correctly received back in the RIO2 board.

5.5 Conclusions

The Event Filter sub-farm High Level Design has been successfully implemented on an SMP machine. To better exploit the hardware resources, a single process multi-threaded implementation has been chosen for the sub-farm code.

The robustness of the software architecture has been checked by performing many tests with varying running conditions. The global throughput achieved is independent of the number of processing tasks and is inversely proportional to the processing time. The load balancing among the processing tasks is excellent (a by-product of the OS scheduler itself) and is not affected by the relative composition in the number and type of processing tasks. The scalability test proves that the performances scale according to the number of processors (more work is needed to assess the compatibility of the results with their defined Spec figures).

All the results obtained until now are a good indication that the scalability is assured and that the software and hardware architectures do not limit the behaviour of the sub-farm.

5.6 References

13 "Proposal for an Event Filter Prototype based on a Symmetric Multi Processor architecture", ATLAS DAQ-1 Note 82, http://atddoc.cern.ch/Atlas/Notes/082/Note082-1.html

14 "Detailed Design and Implementation of an SMP Event Filter Sub-farm", ATLAS DAQ-1 Note 128, http://atddoc.cern.ch/Atlas/Notes/128/Note128-1.html

15 Hewlett-Packard servers, http://www.unixsolutions.hp.com/products/servers/index.html

16 CILEA, Consorzio Interuniversitario per L'Elaborazione Automatica, Milan, http://www.cilea.it

Chapter 6
INTEL Commodity Multi-Processor Approach

The main goal of the THOR project at the University of Alberta is to provide a commodity component prototype for the event filter that may be used to study various algorithms and implementations of the data flow and analysis. The total required computing power for the final ATLAS event filter is estimated to be on the order of 250,000 SPECint95 or about 1000 processors. THOR aims to provide a 1/10 size (128 processor) prototype of this farm in its final version which is planned for the end of 2000.

The current THOR prototype, shown in Figure 27, consists of twenty dual PII/III 450 MHz SMP machines connected with fast ethernet. Each machine has 256MB of RAM and a 4GB IDE hard disk. In addition, nine of the nodes are also connected using Scalable Coherent Interconnect (SCI). A total of 100 GB of disk is available to the cluster, and a 500GB tape robot is also available. An additional dual Pentium machine is used as a firewall gateway in order to isolate the cluster from the external network. This machine also simulates the output of the Event Builder during Event Filter testing. The remaining machines can be configured in many ways, in order to simulate one or many sub-farms of the ATLAS Event Filter. Most often, one machine acts as the farm input and output and the remaining machines run processing tasks (PT). The THOR cluster uses a mixture of RedHat Linux 5.2 and 6.0 as the operating system with kernel version 2.2.2.

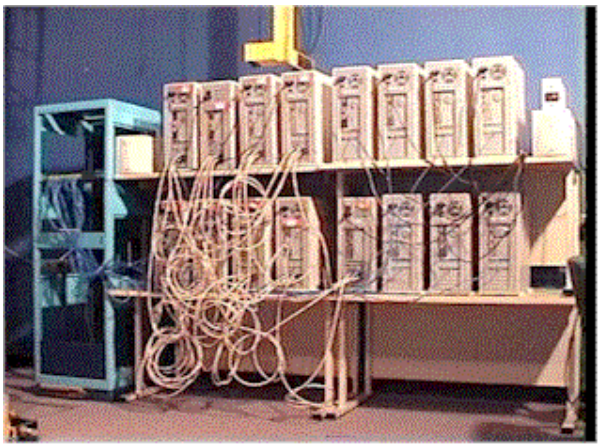


Figure 27 The THOR cluster

6.1 Event Filter Implementation

This implementation of the Event Filter is based closely on the high-level design outlined in ATLAS DAQ Note 61. In particular, the design is based on a farm of independent processors and thus multiple processes running on the different machines in the farm. The primary differences between the work described in chapter 4 and this approach are (a) the operating system chosen was Linux (as opposed to Windows NT), (b) the inter-process communication transport was chosen to be raw TCP sockets (as opposed to ILU) and (c) all processes are single (as opposed to multi) threaded. The reasons for, and impact of, these design decisions are discussed more fully below.

6.2 Architecture

Specifically, the Event Filter is composed of some number of sub-farms. Each sub-farm in turn is composed of a Distributor task, a Collector task and one or more Processing tasks. Conceptually (although not necessarily) each of these elements runs on a different physical machine, all of which are interconnected by a networking fabric which supports TCP / IP.

To date, most of the work has been devoted to implementing a single sub-farm, the so-called “vertical slice” through the Event Filter. A limited amount of work on multiple farms has also been carried out.

As mentioned, a “sub-farm” (shown in Figure 28) consists of:

a. A Distributor process, which connects to the Dataflow and receives events from it. Although [17] specifies the possibility of multiple levels of distributor, we have investigated only a single level, which connects directly to processing tasks. The primary function of the Distributor is to send events it receives from the SFI (Switch-Farm-Input) to an appropriate processing task.

b. A Collector process, which connects to the output of the processing tasks. Its function is to collect events which “pass” the selection criteria imposed by the processing tasks, and forward those events to the SFO (Sub-Farm-Output) for archival storage. Again the possibility is left open in [17] for a multi-level Collector - we have implemented only a single level, directly connected to the processing tasks and the SFO.

c. One or more processing tasks. These receive events from the Distributor, process them and make a decision as to whether the event is to be accepted or rejected. Those accepted are passed to the Collector for eventual transfer to permanent storage.

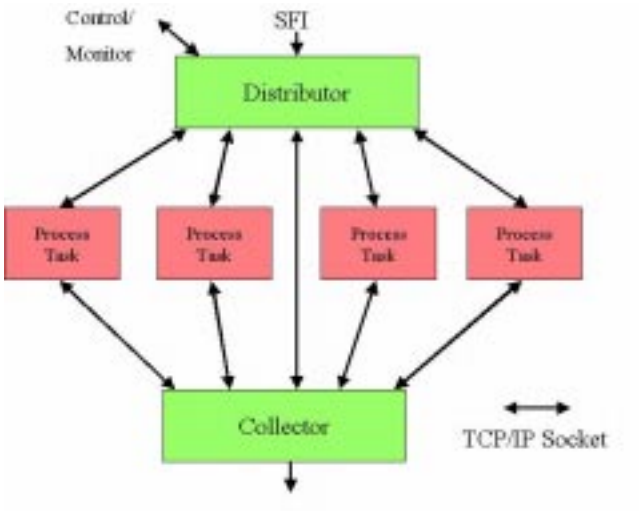


Figure 28 Subfarm architecture

In our initial investigation of multiple farms, we implemented a global “Event Filter Control / Monitoring” task, responsible for control and monitoring of all levels of the Event Filter. It connects only to the Distributors of each farm. In addition to the primary tasks of feeding events to its subsidiary processing tasks, the Distributor must also take on the additional administrative tasks of monitor and control of all processes within its sub-farm. This overall structure is shown in figure 3.

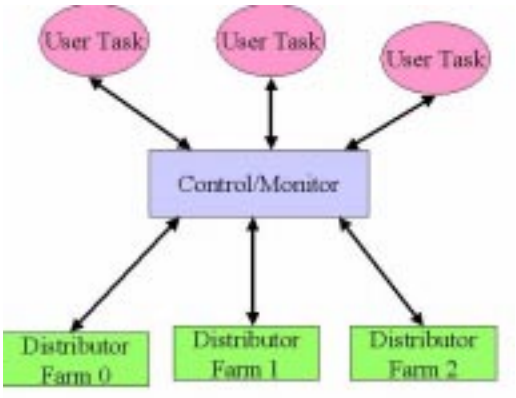


Figure 29 Subfarm implementation

6.3 Operating System

We have chosen to base our implementation on the Open-Source Linux operating system with the 2.2.x kernel series. Although there were a number of reasons for this choice, a major consideration was the ready availability of the system and the fact that (since it is “Open Source”) upgrades and bug fixes are readily available in the public domain. To give a single example, early in the project a problem with the TCP implementation was discovered. Once

the problem was identified, a patch was posted on UseNet within a few days, and a permanent fix was included in the next release of the system (within a few months). It is difficult to imagine a similar level of support from a commercial operating system. An additional reason for our choosing Linux was that it offered the opportunity to directly evaluate the impact of different operating systems on the performance of the Event Filter (c.f. chapter 4). Linux also handles SMP architectures well and is scalable past two processors per node.

6.4 Communications Protocol

We chose to implement inter-process (and inter-machine) communications using raw TCP/IP sockets. There were a number of reasons for this choice. First, TCP/IP is a well-established and well tested protocol. In addition, it is a reliable protocol, which results in substantial simplification of the application code ("reliable" is used here in the restricted sense that the sender of a message is guaranteed to receive either positive notification that the message was received or an error indicating that it was not - no additional user code dealing with verification of message delivery is required). TCP/IP is available on all Unix (and other) operating systems, allowing code developed using it to be easily transferred to different platforms. Finally, at the socket level, a number of options are available to allow the application to tune the parameters of the protocol for optimal performance.

Although all of our work to date has been based on TCP/IP, the interface seen by the application programs hides many of the details. In particular, inter-process communications appear to the application simply as an object called a "connection", which is capable of being read/written by either end. For example, this abstraction should prove useful when we test our prototype using SCI (Scalable Coherent Interconnect) for which TCP/IP drivers are not yet available. An additional goal here will be to evaluate the overheads associated with specific networking protocols.

We have implemented an additional "Event-Filter-Specific" protocol on top of TCP/IP. Each message passed between components is preceded by a header, containing a message type and length, this is required to allow the receiver to segment the byte stream presented by the socket into distinct messages. In addition, each message header contains information on the route which the message has taken (essentially a list of which socket file descriptor was used for each process in the path from sender to receiver). This allows messages to be "forwarded" between components which are not directly connected while still maintaining (within the message) a return path allowing a response to be delivered back to the appropriate source. All of this is maintained in a manner transparent to the application program. Recall that the global Control / Monitor program connects only to the Distributor of each sub-farm. This message forwarding mechanism allows messages destined for the Collector to be sent to the Distributor and forwarded to the Collector (e.g. to retrieve statistics), while automatically maintaining within the message header the return path required for the Collector to return the information to the proper place.

6.5 Details of the Sub-Farm

Each sub-farm consists of the Distributor process, Collector process and one or more processing tasks. A socket connection is established between each processing task and the Distributor (for event input) and between each processing task and the Collector (for event output). Additionally, a direct connection is established between the Distributor and Collector. There are a number of reasons for this. First, the connection of a sub-farm to the Control / Monitoring task is via the Distributor only. The presence of the direct connection allows the Distributor easy access to monitor / control the collector. Second, it will be useful, at least during initial commissioning / testing of the system, to pass a certain fraction of the events directly to archival storage and subject them to more rigorous off-line analysis, to determine that the filter is behaving as expected. Finally, “abnormal” events might cause a processing task to crash - such events must be labelled as such and passed through so that offline analysis can determine why they are “abnormal” and modify the algorithms appropriately.

In addition, the Distributor sets up a socket connection to the Control / Monitoring task and the SFI, and the Collector sets up a connection to the SFO.

When the connection is first established, each side informs the other what kind of process it is. Thus, processing tasks know where to look for input and send output, and, in particular, the Distributor knows which sockets connect to processing tasks and what kind of events they are prepared to accept. Thus, the Distributor has available to it sufficient information to sort events on the basis of type and send them to an appropriate processing task.

Most communication between tasks is based on the “Ready” mechanism discussed in Note 61, in which each component which is to read information from another first informs it that it is ready to do so. The intent is to prevent the writer from blocking unnecessarily when the other side is not yet ready. In practice, this is less important than one might think when using the socket-based implementation. In the interest of simplicity (discussed in more detail below) processing tasks use the ready mechanism only for the connection to the Distributor - they signal ready and then wait for an event to be sent. The Collector, on the other hand, simply waits (via the ‘select’ system call) for a processor to indicate that it has written something and then reads it - the processor will thus block when it writes if the collector is not yet ready to read. This is not a large performance impediment if the number of processing tasks is large enough. This ready mechanism is used only for event transfer (assumed to be a large amount of data). It is assumed that the communications protocol (TCP / IP in this case) and / or operating system provide sufficient buffering that small messages can be sent asynchronously. This is in fact implicit in the Ready mechanism anyway - there must be sufficient system resources to receive the ready message at any time. The Distributor / SFI and Collector / SFO connections both use the ready mechanism.

A primary design goal in our implementation is that processing tasks should be kept simple (at least as regards their interaction with the rest of the Event Filter software). It is assumed that the number of different types of processing tasks may be large (and dynamic, as different Physics goals are identified). The tasks of the Distributor and Collector, on the other hand, can probably be specified relatively completely “once-and-for-all”, and so any complexity should properly be included in these tasks (which are more likely to be written once).

The system has been designed to be robust and fault-tolerant. In particular, any process (except the Distributor) can fail and not have an adverse effect on the remainder of the sub-farm (such protection for the failure of the distributor could easily be included, at the cost of additional complexity in processing tasks). Additionally, processing tasks can be added dynamically as the need arises - they connect seamlessly to the Distributor and are available to receive events to process.

6.6 Integration

The Event Filter prototype has been successfully integrated into the dataflow and tested by communicating with the DAQ prototype at CERN in Building 40. Events were passed successfully to Alberta where they were analysed by the processing tasks in the prototype before being returned to CERN. Clearly this integration test was aimed at establishing the ease of the integration procedure rather than making performance measurements. The Alberta EF code also contains all the required hooks to be compatible with the backend as specified in version three of the Event Filter API.

6.7 System Tests

Several throughput tests of a single sub-farm have been performed in order to illustrate the operation of the THOR prototype. In order to test the network bandwidth, an initial test was performed using 1 MB simulated ATLAS events and turning off the processing in the processing tasks. The sub-farm consisted of the injector, distributor, collector, and injector running on a single SFI/O machine, and two processing tasks running on each of five PT machines for a total of 10 processing tasks. This resulted in a total throughput of 7.94 events per second showing that close to the entire bandwidth of the 100 Mb/s full duplex network was being utilized. The calorimeter processing task requires 0.14 s per event to complete on the Pentium II 450 MHz processors used in THOR. It is expected that processing a single event will take on the order of one second on the processors in the final event filter model, so the processing tasks in the THOR prototype were set to loop over the calorimetry task seven times in order to simulate this processing time.

With the processing loop set to seven iterations, a scaling test of the event throughput was performed. The injector, distributor, collector, and ejector were run on the SFI/O machine, and runs were performed with increasing numbers of processing tasks to a maximum of two per PT machine. The results of this test, shown in Figure 4, reveal a constant increase in throughput with the number of processing tasks. The higher slope in the throughput curve between two and three processing tasks reflects the fact that two nodes (and hence two Ethernet cards) are used for the three processor case while only one Ethernet card in one node is used for the two processor case. In order to study the crossover between network and processing limitations to event throughput, the number of loops through the processing tasks was increased and the resulting throughput measured. The farm was configured with the injector, distributor, collector, and ejector running on the SFI/O machine and two processing tasks on each of the five PT machines for this test. Two event sizes, 0.25 MB and 1.0 MB were used in separate tests at each of 1, 3, 10, 30, and 60 loops through the processing per event. As

can be seen in Figure 5, the processing bound is reached much more quickly with the smaller event size. The network bound region can be seen as the short flat section at the beginning of the 1.0 MB event size curve. The fault tolerance of the prototype was tested by killing and restarting processing tasks by hand during a run. In all cases, the killing of a processing task was handled gracefully and no events were lost. Restarted processing tasks were able to join the farm immediately and began to process events.

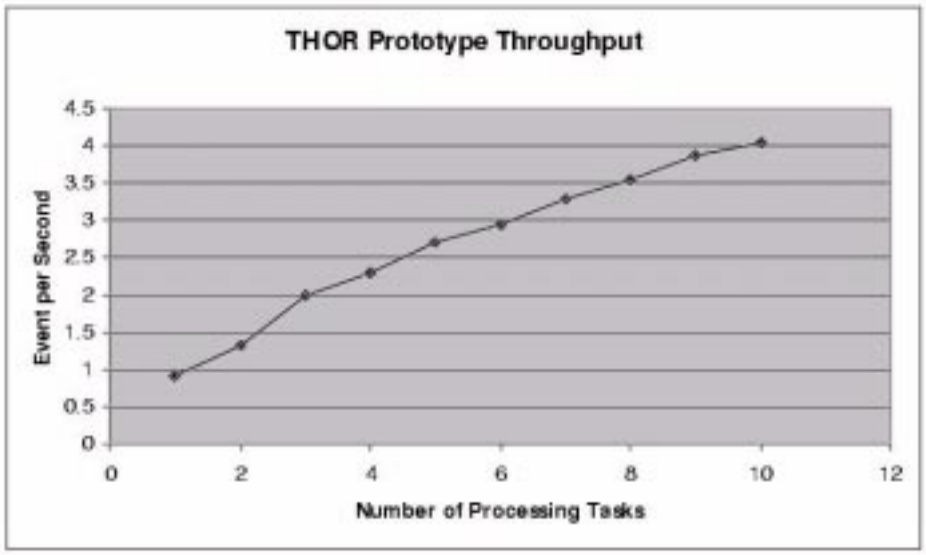


Figure 30 THOR prototype throughput as a function of the number of processing tasks

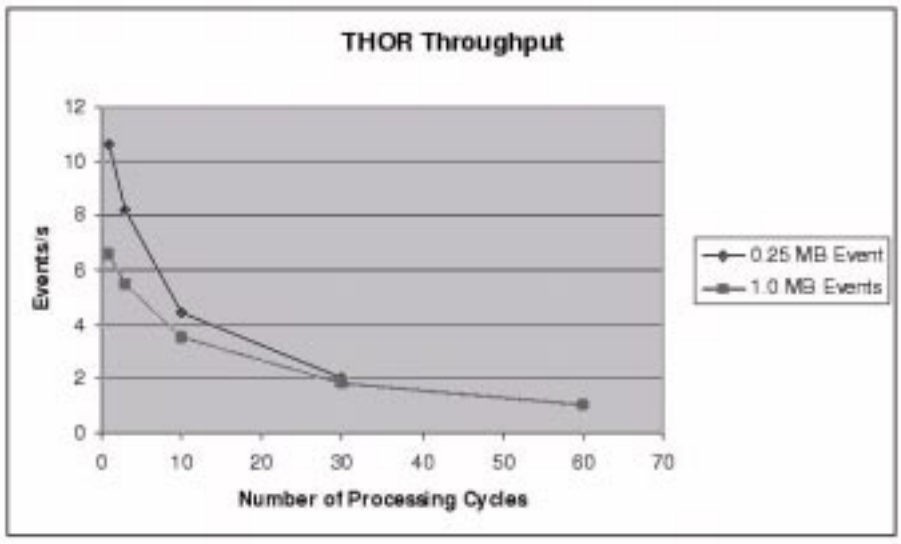


Figure 31 THOR prototype throughput as a function of the processing time

6.8 Future Plans:

Plans for the near future of the THOR prototype and the Alberta Event Filter implementation include acquiring and testing the algorithms on quad pentium computers, implementing SCI as a network transport using the low level SCI API, implementing multi-thread support for the distributor and collector, and improving the monitoring capabilities. We also hope to investigate multiple sub-farms more fully and to perform a direct comparison with the Marseille implementation (chapter 4). The THOR prototype itself will double in size from 40 to 80 processors in the near future with 36 processors being connected by Scalable Coherent Interconnect. Gigabit Ethernet is also being evaluated as a possible networking solution

6.9 References

17 A High Level Design of the Sub-Farm Event Handler, ATLAS DAQ Prototype-1 Technical Note 61, http://atddoc.cern.ch/Atlas/Notes/061/Note061-1.html

Chapter 7
Comparison of the prototypes

The previous chapters have described the available results from the three prototypes, in terms of implementation of sub-farm software, throughput, scalability and robustness etc. Of course, work will be continued and extended, based on these results, in order to achieve the best possible knowledge of the behaviour of an Event Filter sub-farm. In the following, we will attempt to identify some of the crucial aspects which highlight the benefits of each solution, including all developments belonging to a particular implementation, to guide the reader in a comparison of the different prototypes.

7.1 Data redundancy and robustness

The prototypes implement a form of the Distributor Global Buffer, as described in the High Level Design. In all implementations this currently takes the form of a disk partition, where events are stored during their passage through the sub-farm. This is important in order to be able to recover from a crash of the processing task or of one Distributor, or even of the sub-farm itself. As noted elsewhere, in principle, another solution is to rely on the core-dump function of the operating system itself. Leaving this solution aside (which does not anyway solve the problem of the sub-farm crash itself), one should note that, as seen from the measurements in all the prototypes, the time spent in disk writing is important only if the processing task time is smaller or comparable to it. With the prospected speed of disks, or disk-arrays, compared to the ATLAS reconstruction and analysis time, one could say that this item should not be a limitation in the performance of the sub-farm. On the other hand, since an error-free sub-farm is clearly not achievable, a better estimate of the tolerable, unbiased data loss rate for the ATLAS detector should be given in the future.

7.2 Data communication mechanisms

Several mechanisms have been studied in the prototypes to send data to the various processing tasks on different processors. In the distributed implementations (Marseille and Alberta) this is achieved via either CORBA / ILU or TCP structures.The interesting advantage of the TCP solution is that it is lighter than the ILU one, and hopefully general enough for the needs of an Event Filter sub-farm. The least demanding solution in term of resources is the

one adopted for the SMP sub-farm, where the computer memory is used to hold the events and hence pointers, passing event addresses to the processing threads, are the only communication mechanism used. Given the RAM costs, this should not constitute a problem for reasonably sized sub-farms, even though the hardware of the sub-farm will be intrinsically more costly. An interesting alternative solution is represented by the SCI interconnect between processors, currently studied by Alberta, which builds a sub-farm of the cc-NUMA type, using commodity components. One might note that, in all the latter examples (with respect to CORBA/ILU), some more work is needed to accomplish the correct identification of objects which need to be known in the outside world (e.g. for Back-End software purposes).

7.3 Throughput and scalability

The output of the ATLAS Event Builder, after Level-2 selection, is currently estimated at about 1 KHz. From this number, together with estimates related to the processing time needed for each event, in the ATLAS Technical Proposal several figures have been given, related to the size and the granularity of the Event Filter sub-system. From the measurements obtained by the three prototypes and presented in this document, one could easily infer that no problem is envisaged concerning the fulfilment of the requirements imposed by the ATLAS DAQ system. In fact, all sub-farms are currently achieving, or are close to achieving, throughputs consistent with the required ATLAS numbers. This, of course, assumes that the EF will be able to process and analyse a full ATLAS event on average in 1 second. Moreover, complex scenarios arise from the mixture of physics and non-physics data (e.g. calibration data) flowing through the farm. In this case, the current understanding is that load balancing of events within and between sub-farms will be a key issue, if one wants to merge those data types in the same sub-farm.

In the scalability studies, very interesting performances have been presented. In the SMP case, where the use of kernel threads implies relying heavily on the Posix operating system structure, the prototype has been shown to scale up to several tens of processors and hundreds of processing tasks. In the distributed architectures, assuming the average event size of 1 MByte, the use of new generation commercial networking should be capable of ensuring the necessary performances. A particular role is played by the THOR prototype, where the scalability studies are extending beyond the scope of “DAQ/EF -1” Prototype, addressing the problem of multiple sub-farms and their partitioning.

7.4 Sub-farm configuration and monitoring

Configuring the sub-farm and keeping track of its behaviour is an essential ingredient which all the prototypes have considered in their studies. In the Marseille implementation a lot of progress as been made in understanding the supervisor element, based on Java Voyager. This is particularly relevant to demonstrate the feasibility of a complex distributed architecture, where bottlenecks might arise from the inability of communicating to and keeping control of thousands of processors. The achieved results are extremely positive and are of course beneficial also to other implementations. The monitoring component has been studied in the same context, too, providing a graphical interface to the sub-farm performances, which

extends for THOR across many sub-farms. In the SMP case, internal monitoring is provided through proprietary tools which control the behaviour of the processing tasks, whilst for general supervision a tool such as the one developed in Marseille could be adopted.

7.5 Architectures and costs

In the spirit of comparing and confronting different architectural solutions satisfying the requirements of the ATLAS Event Filter sub-system, we have tackled the problem using distinctive points of view. On one side, the use of low-cost hardware lead to the studies of widely distributed architectures. On the other side, the quest for a robust, self-consistent solution based on the adoption of commercial SMPs. In the outside world, we see many examples of similar architectures, in particular big computation centres are more and more based on PC-like clusters, whilst mission critical data-driven companies tend to adopt proprietary multi-processor systems. In evaluating the cost of each solution, one should bear in mind that an Event Filter sub-system will need to last several (~15) years, hence the cost is not merely limited to the buying of the hardware. Maintenance of the entire system is an issue which has to be addressed as well to evaluate properly the cost, including software and hardware upgrades, mean time between failures, hardware modifications, operating system issues and, of course, the associated manpower. It is not the goal of this document to propose one particular solution, or to take decisions on the best architecture. The success of the PC-like solutions, anyway, backed-up by the presence of performant SMP-compliant operating systems, with thread-based kernel, suggests a very promising implementation based on multi-processors machines built around Intel chips of the new IA-64 generation. This will allow for a low-cost, reasonably distributed architecture for the Event Filter. Currently we see on the market Intel SMP boxes with up to 8 processors. Many joint ventures and partnerships exist in the commercial world for the new chip and this will certainly be beneficial for us, on the right time scale. It should be noted, in conclusion, that, at the present stage, even for the whole farm there are two approaches that look equally appealing. One gathers processing power by building a self-made architecture around small boxes and the other tries to exploit commercial solutions and partnerships to fulfil the Event Filter goals. We will actively pursue investigations in both areas, to come up with the best, most performant and robust solution for ATLAS.

Chapter 8
Outlook

In the previous chapters we have presented the current status of the studies performed until now in the Event Filter area. The results shown are a clear indication of the deep level of understanding reached in the implementation of an Event Filter sub-farm, but of course are far from being final. There are many issues which deserve a deeper level of investigation, in order to reach a comprehensive knowledge of all the parameters which regulate the behaviour of the sub-farm of the entire filtering machinery. In fact, in many areas, one should now try to enlarge the scope of the analysis towards the global Event Filter system.

In the following, a few guidelines for the next development phase are presented, with the aim of being reasonably representative of the work to be done in the near future, after the publication of the Technical Proposal for Data Acquisition, High Level Triggers and Detector Control System. It is also sensible to assume that the relative priorities and even the list itself are subject to any change which will derive from an improved understanding of the sub-system and its correlations with the ATLAS Trigger / DAQ.

- study the configuration aspects of the sub-farm, including Operating System issues (like patches, updates, etc) and their correlation with other parts of ATLAS DAQ (notably Back-End software and configuration databases)
- improve the supervision and monitoring of each sub-farm and of the Event Filter sub-system as a whole and identify the means to transfer this information to the reporting lines of the DAQ system
- study a multi-process implementation on SMP boards and compare its performances and behaviour (in terms of load-balancing, error handling, robustness, flexibility) with the multi-thread solution
- assess the interplay between the flowing of physics data and non-physics data (calibration, monitoring, etc) in the farm and derive information of the eventual partitioning (time dependent) of the Event Filter sub-system
- study the best means to access off-line calibration and alignment databases (and the associated versioning), and the implication of making those data available to thousands of nodes, in particular for the widely distributed architecture
- identify, in collaboration with the Physics and Event Selection Algorithms (PESA) group, a complete selection and classification strategy for the Event Filter, in order to achieve the best possible rejection whilst keeping the highest physics discovery potential

- use the knowledge gained in the studies of the PESA group, and pursue further investigations in order to derive the level of flexibility needed in the High Level Trigger sector and understand how to map it onto different hardware implementations (moving selection algorithms and functionalities)
- monitor the development of the new ATLAS Object Oriented reconstruction software and make sure that this suits the Event Filter needs in terms of architecture, performances and robustness
- build complete error handling procedures capable of keeping the data loss at a very low level (to be agreed with the ATLAS community) and of handling failures in the whole system (restart components, kill unwanted processes, etc)
- track the technology development and its industrial strength to proceed towards a better assessment of the Event Filter Farm global architecture and final implementation.
- evaluate the impact of non-homogeneous hardware/software implementations on the performances and operability of the Event Filter sub-system and investigate tools to handle these conditions

These ideas (and all the others not listed above) should be clearly mapped in a functionally organized way, in order to assign correctly responsibilities and tasks. Moreover, whenever possible, ATLAS should try and profit from the expertise of other experiments (in particular the LHC ones) and the knowledge of the IT Division at CERN.