Summary Document of the Event Building Studies within the DAQ-1/EF -1 project

ATL-DAQ-2000-048
04/08/2000

Authors : **G. Ambrosini, E. Arik, H.P. Beck, S. Cetin, T. Conka, A. Fernandes, D. Francis, Y. Hasagawa, M. Joos, G. Lehmann, J. Lopez, A. Mailov, L. Mapelli, G. Mornacchi, Y. Nagasaka, M. Niculescu, K. Nurden, J. Petersen, D. Prigent, J. Rochez, R. Spiwoks, L. Tremblet, G. Unel, S. Veneziano, Y. Yasu**

Keywords : Eventbuilding, Data acquisition, Dataflow management, Networking, Switch, Ethernet, ATM, Simulation, Ptolemy

Abstract

The Event Building sub-system of the DAQ/EF -1 prototype is introduced from its design up to its implementation. Performance measurements on a small-scale prototype using two different networking technologies are compared with simulation results obtained via a computer model based on Ptolemy. The results obtained from these studies are promising for reaching the Atlas requirements but need to be confirmed on bigger prototype systems.

1 Introduction

1.1 Purpose of the document

This document summarises the work performed, within the context of the Event Builder of the DataFlow system in ATLAS DAQ/EF prototype -1 [1][2].

1.2 Overview of the document

The document consists of six sections. In section 2 the design of the DAQ/EF-1 Event Builder is described. This is followed by a description of the baseline event builder configuration and of the measurements performed on the ATM setup and the Gigabit Ethernet setup. The work done in the area of modelling and simulation is described in section 4. Reliability and fault tolerance are described in section 5. Conclusions are presented in section 6.

1.3 The Event Builder in DAQ/EF-1

The Event Building sub-system is an integral part of the DAQ/EF prototype -1. It provides *the coordinated, concurrent transfer of different sets of event fragments to different destinations.* It allows one or more sub-detectors (or partial sub-detector[1]) to operate in stand-alone mode[2], while all other sub-detectors participate to collective Event Building. In addition, it allows events of a specific type, defined by a system wide attribute, to be built at a set of unique destinations. This is shown schematically in Figure 1:

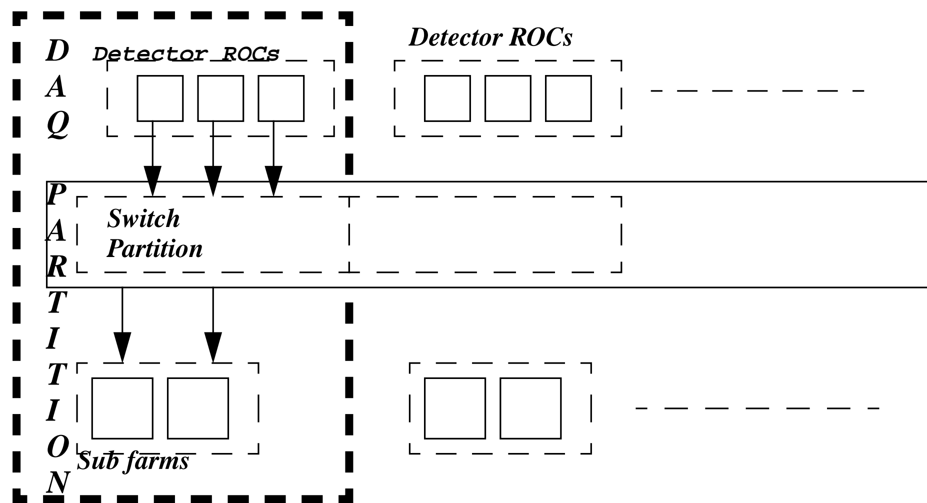

Figure 1: Schematic of Event Building in DAQ prototype -1.

1.3.1 General Requirements

The high level model for the DAQ prototype -1 Event Builder is formulated independently of any given communication technology, therefore it should be capable to incorporate different communication technologies.

1. A partial sub-detector is a sub-set of sub-detector ROCs.
2. The events are built from a sub-detectors event fragments independently of other sub-detectors.

The problem of merging event fragments into a complete event is characterised by the following constraints:

- An event is defined by: a global event identifier (Gid), an event type (Etype) and the partition identifier (Pid).
- An event fragment is defined by: a global event identifier (Gid), an event type (Etype) and a source identifier (Sid).These parameters will be used to determine the set of destinations of the event fragment.
- Each event fragment is transferred to one and only one destination.
- The number of sources that send event fragments from the same event to the same (set of) destinations does not change during the run. A (physical) partition is statically defined at run start time.
- The number of sources that send event fragments to the same set of destinations depends only on the partition sources and destinations belong to.
- A destination assigned to a particular event shall be able to receive that event.
- Error conditions are detected, flagged and reported. The occurrence of an error condition may lead to loss of event fragments.

2 The Design of the Event Builder

In this chapter the design of the Event Builder is described. After an overview of the high level design and protocol, the detailed design for its main elements will be outlined.

2.1 High Level Design of the Event Builder

The Event Builder (EB) is a sub-system of the DataFlow in the DAQ prototype -1 and is responsible for merging fragments coming from different parts of the detector to full, formatted events. The EB subsystem and its interfaces to other components of the Trigger/DAQ is depicted in Figure 2.

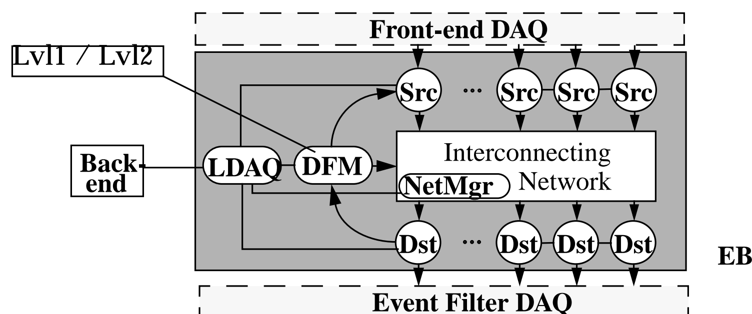

Figure 2: The dataflow system.

The EB introduces the concept of physical partitions in the data acquisition system allowing for concurrent disjunct data taking sessions for sub-detectors (or partial sub-detectors[1]). In

1. A partial sub-detector is a sub-set of sub-detector ROCs.

addition, it allows events of a specific type, defined by a system wide attribute, to be built at a set of unique destinations (the so called logical partitioning).

The Event Builder model [3] consists of logical objects and a high-level protocol. Five logical objects have been identified: the Source (Src), the Destination (Dst), the Data Flow Manager (DFM), the Network Manager (NetMgr) and the Network. The high level protocol is a set of rules which defines how events are built. It consists of control mechanisms, to manage the flow of data, and of a transfer mechanism. Source and Destination use the high level protocol rules to build the event, while the DFM provides the control rules. The transfer of event fragments as well as of control messages occurs over a Network which is configured and controlled by the Network Manager. The relationships between the first three logical objects are shown in Figure 3: whereas the time ordered sequence of the high level protocol is depicted and Figure 4.

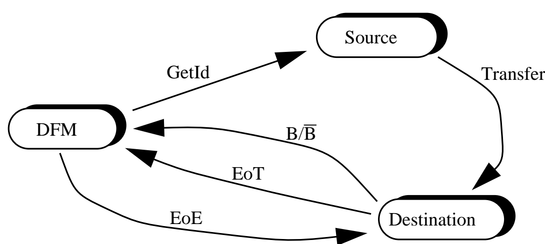

Figure 3: Event Builder high level protocol. Object relationships.

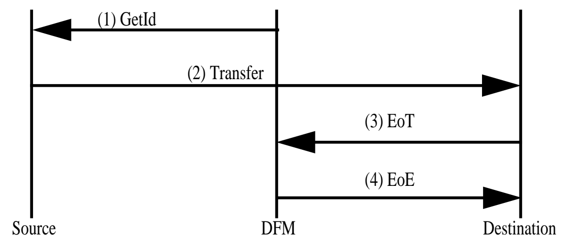

Figure 4: Event Builder high level protocol. Time ordered sequence.

The set of mechanisms defining the high level protocol is: *GetId*, *Busy*/$\overline{Busy}$, *End-of-Transfer* (EOT), *End-Of-Event* (EOE) and *Transfer*.

When an event is scheduled for Event Building by the Trigger system (Lvl2 or Lvl1) the DFM uses a *GetId* mechanism to inform the source on the Destination Id (Did) and the Gid[1] of an event fragment which has to be exchanged with the destination. The actual event fragment exchange is performed by a Transfer mechanism. The destination notifies the DFM, via the End-of-Transfer mechanism, when an event fragment has been received. The End-of-Event mechanism is used by the DFM to inform the destination that a complete event has been exchanged between the destination and sources. In the case where a destination becomes "busy", a *Busy*/$\overline{Busy}$ (B/$\overline{\text{B}}$) mechanism is used, per event, between the destination and the DFM to halt the flow of event fragments to the destination. This mechanism is also used to resume the flow of event fragments.

The Source

The source is a component of an I/O module. It is technology invariant; the technology specific layer is provided by the network interface. The source provides the following functionality: initialisation of the network interface, event fragment sending and the handling of errors related to either sending or to the network interface.

1. Gid = Global Id. This identifier may coincide with the bunch crossing Id or may be an extension of it in order to be a unique identifier across a whole data taking session.

The source exchanges an event fragment with a destination. The Gid of an event fragment and the Did of the destination are obtained from the DFM. The source uses the Gid to access the event fragment buffer and, with the Did, activates the network interface. The latter subsequently performs the send of the event fragment. The source subsequently performs those actions necessary to complete the event fragment exchange, like, e.g., releasing the buffer containing the fragment.

The Destination

The destination is a component of an I/O module. It is technology invariant, the technology specific layer is provided by the network interface. The destination provides the following functionality: initialisation of the network interface, event fragment “ordering”, event assembly and the handling of errors related to any of the former.

The destination receives event fragments via the interconnecting network, notifies the DFM each time a fragment transfer has been carried out and is informed by the DFM when a whole event has been (successfully or not) completed. The Dst must be able to send a busy signal to the DFM if there is not enough space in the buffer to accept all the fragments belonging to a new (not yet assigned) event. Furthermore, the destination performs the event assembly and provides the event header and the sub-detector headers. The copy of the assembled data to a complete event in contiguous memory space is not a task of the Dst.

The DFM

The DFM is the logical object which ensures the correct flow of event fragments between sources and destinations. It therefore defines the high level protocol control rules. It assigns Dids to event fragments and traces the reception of event fragments at destinations.

The Did assignment policy is based on a set of attributes which define the statical partitioning of the system. The physical partition Id uniquely identifies a set of sources and destinations, the Event type can force some events to be sent to particular destinations or to include only some sources in the Event Building process.

The assignment policy selects a Did from a possible set of Did’s. While the association of a set of Dids with a set of Sids is static and performed at initialisation the final selection of a particular Did within the set to pair with a Gid is done dynamically on a per event basis.

For each Gid and Did pairing the DFM must record which sources have exchanged an event fragment with the destination. When all sources (statically set at initialisation) have exchanged event fragments, the DFM informs the appropriate destination that the event is complete. The DFM may also inform a destination that an event is incomplete when it deems that one or more sources, based on a pre-defined criteria, will not complete the exchange of an event fragment.

The Network Manager

The Network Manager (NetMgr) watches over the status of the EB data flow network and provides information about the status of the other elements in the EB system. It has, in particular, to ensure the correct initialisation of the EB data flow network before the data flow tasks (like EB sources, EB destinations and DFM) can use it.

The Network

The Network comprises all those elements which are needed for the movement of event fragments and control data within the Event Builder. It consists of Network Interface Cards, a switching network for data transfer and, possibly, a separate messaging system for control messages. While all other logical elements can be designed and implemented independently from the communication technology, the network is by definition technology dependent.

2.2 The Data Flow Manager

The DFM is the logical object which ensures the correct flow of event fragments between sources and destinations. It assigns destination identifiers (Dids) to event fragments and traces the reception of event fragments at destinations.

2.2.1 Interfaces

The DFM interacts with the Source and Destination elements of the EB. The interface is given by the Network interface. Furthermore, it interacts with the LDAQ component of the Data-Flow which provides the interface to the Back-End subsystem for configuration, run control and error reporting. The Event Builder foresees the introduction of a ‘SUPER-LDAQ’ which will create an instance of the DFM for each physical partition in the system. The DFM will then control the partition it was assigned to and exchange control commands with a DFM-LDAQ. Finally the DFM interacts with the Trigger System in order to be informed on which events are scheduled for event building and to possibly activate a back-pressure mechanism in case that the EB cannot handle new events.

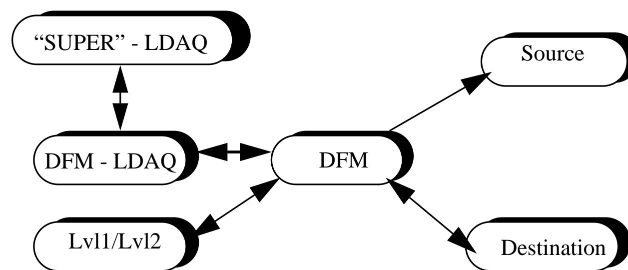

Figure 5: DFM context Diagram.

2.2.2 Functional Decomposition

The DFM is responsible for the correct operation of the EB. On the basis of its high level design and of its context diagram a set of logical components can be identified, which perform the required functions and their interactions. This decomposition involves as well a more detailed specification of the functions in terms of their (input/output) parameters and the data structures they share. A possible design, in which the physical partitioning is handled by the ‘SUPER’-LDAQ, is shown schematically in Figure 6.

The global control (‘SUPER’-LDAQ)

The global Control of the event building system is the task of the so-called ‘SUPER-LDAQ’. It is responsible for running and controlling all the possible physical partitions in the system. It

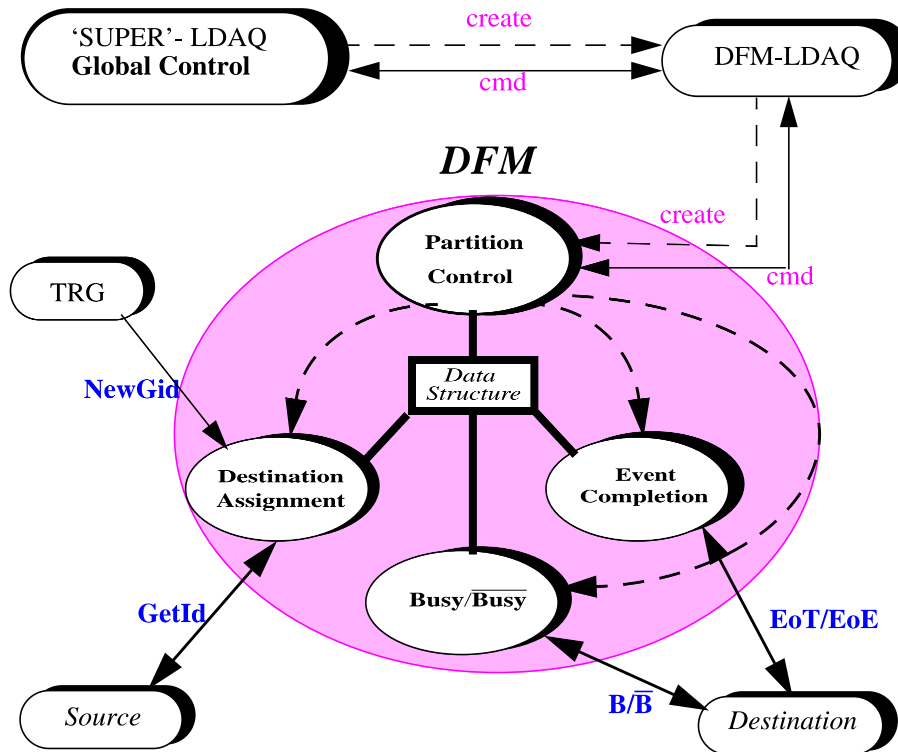

Figure 6: Schematic of the functional components of the DFM and their interaction.

achieves that by creating and concurrently running instances of DFMs with associated LDAQs and by keeping track of their current status. Commands which activate and control the single partitions (e.g. pre-configure, configure, start, stop, etc.) are conveyed to the DFMs via the DFM-LDAQs.

The partition control (DFM)

It controls all the other components of the DFM by transmitting the commands and database information from the DFM-LDAQ. Commands, the same set to which any DAQ/EF -1 I/O Module will respond, accepted by the partition control module are the following:

boot: initial operations to set up the partition;

pre-configure: it initializes the data structures used by the DFM components and performs first setup operations for high level protocol functions;

configure: it accomplishes the necessary initialization phase in order to be able to perform the high level protocol functions;

start: it activates the partition;

stop: it halts the execution of the partition;

pause/resume: it suspends/resumes the partition;

unconfigure: it deactivates the partition;

exit: it aborts the partition and quits.

The destination assignment

This DFM component has the task to associate a certain Did for a given set of event fragments. It receives data input pairs (Gid, Etype) from a trigger system and assigns a destination identifier to it. The pairs of event identifier and destination identifier are then multicast to the relevant sources of the partition.

The assignment policy has basically two steps:

- Identify the set of Dids associated to the partition and event type.
- Assign a Did from the set.

This assignment policy allows partial, full or stand alone event building functionality to be implemented. Any physical or logical partitioning of the EB is taken into account during step one. The second step of the policy implements the real destination assignment algorithm (e.g. round robin or random assignment). Exceptional conditions (such as the “busy” state for a destination) are treated at the second step of the assignment procedure where a suitable Did can be chosen within the available set.

The set of destinations can possibly vary during execution, due to the occurrence of events such as the failure of destinations. The DFM should therefore be able to keep track of what happens during a run and be able to dynamically ‘adapt’ its behaviour in response to the occurrence of unpredictable events.

The Event Completion server

It implements the EoT and EoE mechanisms between DFM and destinations.

The EoT mechanism is used to count the number of fragments received by a given destination, while the EoE mechanism asserts the completion of the event at the destination. The parameters involved in this high level functions are the event Gid, Sid and Did.

The DFM deals with Gids, Sids and Dids in order to accomplish the EoT mechanism. As soon as it detects the partial or total completion of an event (EoE), the Gid-Did parameters along with the status of the completion are conveyed to the destination.

The Busy/$\overline{\text{Busy}}$

The DFM must be aware of the fact that a (set of) destination(s) could temporarily be unable to receive further data. In reaction to that the DFM should halt the transfer of event fragments towards that particular (set of) destination(s), by temporarily changing the assignment policy. Analogously, the DFM must be informed about the recovery of the destination(s). The DFM uses this mechanism to decide, before assigning any Did, whether that particular destination will be able to receive and process the event (Gid) to be transmitted. The purpose is to guarantee the availability of the resources at the destinations at any time during the data transmission. This mechanism can be accomplished by through a credit based Did assignment policy and an additional exchange of information between DFM and Destinations. The flow of new events to a Dst will be halted both in case that its credit is exhausted and in case that it actively requires

to be excluded from new assignments (e.g. in case of the Event Filter SubFarm being unable to process other events).

The partition data structures

There are two kinds of data structures which are used by the DFM components. The first kind is mainly handled by the partition control task and represents the status of the process execution (initialised, running, paused, stopped, aborted, etc.); the second kind of data structure is shared among all other DFM components (the destination assignment task, the event completion server and the Busy/$\overline{\text{Busy}}$) in order to regulate the high level protocol functions.

The partition description contains the following information:

- physical partition:
    - set of sources by Sid and (physical/network) address;
    - set of destinations by Did and (physical/network) address;
- logical partition:
    - Did vs Etype association

The data structures used to control the correct execution of the high level protocol mechanisms have the following information:

- Status at the destinations
    - active/inactive
    - Busy/$\overline{\text{Busy}}$
- Event association (Gid, Did and the number of outstanding EOTs) for each event at the destination.
  A matrix keeps track of the status of the assignment at any time: for each destination there is a set of Gids assigned to it. The possible maximum number of Gids depends on the number of concurrent events that can be simultaneously assembled by the destination and defines what it is called *resource index f*or a given destination. This matrix is updated either by assigning a destination to a new event or after the completion of an event (EoE). The event association follows a credit based scheme. The latter is based on the estimation of the maximum number of events that can be simultaneously received and assembled at the destination and might take into account features of the host architecture (I/O and memory) at the destination.

2.2.3 DFM Implementation

The DFM is an application driven by stimuli which may be generated either externally, by interacting with the external components of the system (e.g. trigger, sources and destinations), or internally, by software (e.g. a timeout expiration). It has been structured as a set of logical components, whose software implementation is called thread or task. These tasks are executed on the occurrence of stimuli and provide the required functionality associated with that stimulus. They are controlled by a scheduler whose scheme is the same as the one of the IOMs.

Based on the analysis of the DFM functionality presented above the following tasks or functions can be identified:

- The DFM-Trigger communication task.
  It provides the functionality to notify the DFM about the availability of a new event to be built
- The DFM-Source communication task.
  It is used to distribute the paring (Gid, Did) to the sources.
- The DFM-Destination communication task.
  It provides the mechanism for controlling the event completion and the availability of resources at the destination.
- The LDAQ interface
  This component fulfils all the functions related to the interaction between LDAQ and DFM such as control, monitoring and error reporting.
- The DFM core.
  This component covers all the functions related to the fulfilment of the high level protocol: destination assignment, event completion and availability of resources at the destination.

The skeleton of the DFM reveals therefore five active elements (threads) as schematically represented in Figure 7.

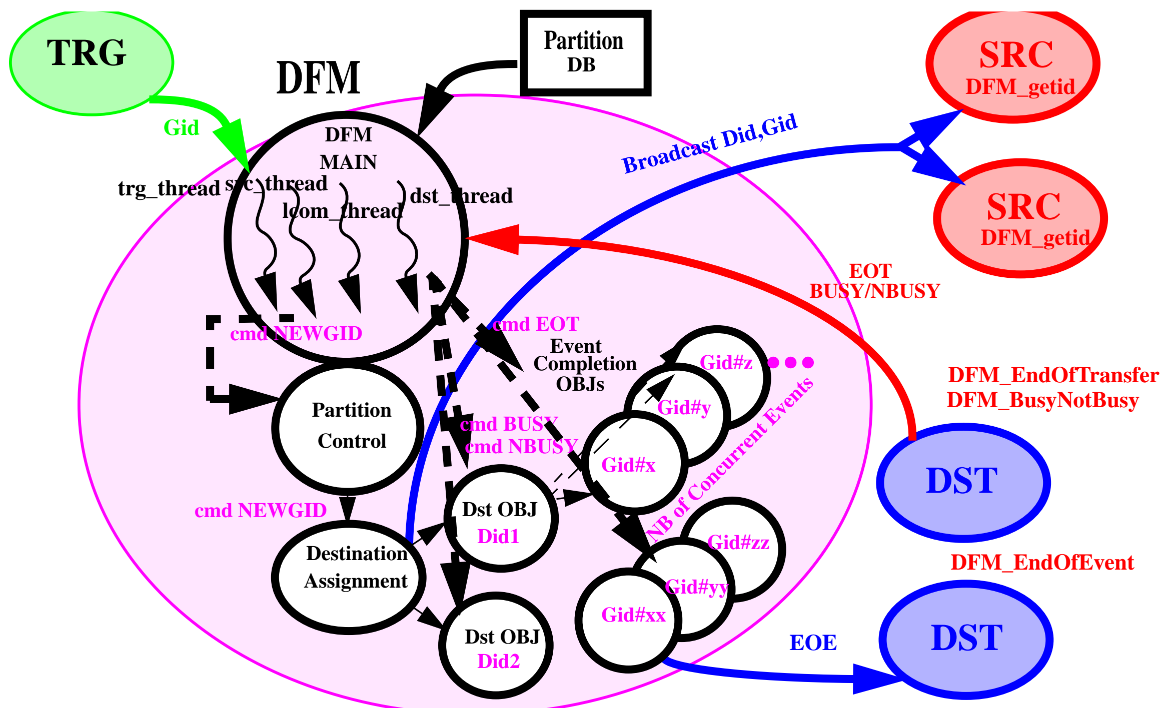

Figure 7: The DFM skeleton.

The core of the DFM functionality has been prototyped and implemented by defining a set of C++ classes which behave as Finite State Machine (FSM) objects.

The DFM/FSM base class provides a method (*getCmd*) to provoke a state transition which, in turn, induces the execution of an action performing the required DFM function associated with that transition. Two matrices are used to define and keep track of state transitions and associated actions. At instantiation time the FSM C++ class creator requires pointers to both the transition state matrix and the action matrix. These matrixes are initialised and assigned at object instantiation. The two matrices define the core of functionality provided by the object.

In addition, the DFM/FSM class provides a method (*sendCmd*) to induce the state transition of a targeted DFM/FSM object so as to allow different DFM/FSM objects to interact with each other.

To cover the functionality required by the logical model described above, it is necessary to define the following DFM/FSM objects: the partition control object, the destination assignment object, the destination object and the event object. A detailed description of their functionality and interaction is given in the section below.

The Partition Control Object

The 'Partition Control' DFM/FSM object executes a set of commands which operate on the status of the DFM and transmits them directly to the object which performs the destination assignment task. It is instantiated when the whole DFM application starts. At the pre-configure phase the destination assignment object is created. It is then initialized at a second step, in response to the configure command, by which the objects handling the event completion at the destination are also created. Once initialized, all the objects belonging to the DFM core are able to receive commands from the 'Partition Control' object (start, stop, pause, resume, abort). The FSM diagram which defines the partition control object is depicted in Figure 8.

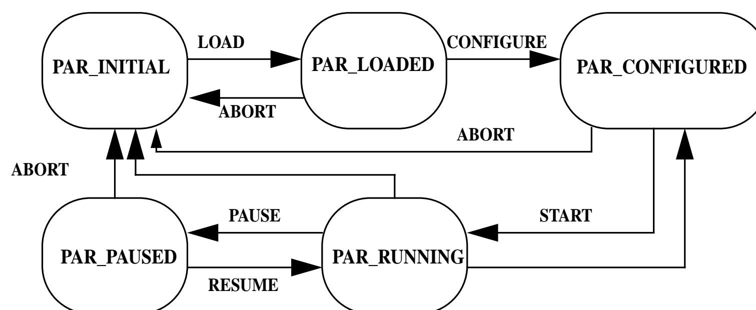

Figure 8: Partition Control FSM diagram.

The Destination Assignment Object

It is instantiated by the 'Partition Control' object and it is responsible for assigning a destination to each set of event fragments (identified by a Gid) notified by the Trigger system. It also interacts with the set of 'Destination' objects which control the event completion and the status at the destination. After a destination (Did) has been assigned to a given Gid, the pair of Did and Gid is broadcast to all sources in the partition. The 'Destination Assignment' DFM/FSM receives commands from the 'Partition Control' object for handling the execution of the partition in addition to a specific command, the PARTITION_GETID. This command is received from the DFM-Trigger communication thread whenever a new set of event fragments is notified by the trigger system. The partition specific commands are the following:

PARTITION_INIT, PARTITION_START, PARTITION_STOP, PARTITION_PAUSE, PARTITION_RESUME.

At initialization a set of ‘Destination’ DFM/FSM objects are instantiated, as many as the number of destinations in the partition. As already mentioned, these components perform all the functions related to the control of the availability of resources at the destination and the event transfer completion. Whenever a new Did has been assigned, the related ‘Destination’ DFM/FSM object is notified by the ‘Destination Assignment’ object via the DESTINATION_GETID command. The FSM diagram of the ‘Destination Assignment’ component is shown in Figure 9.

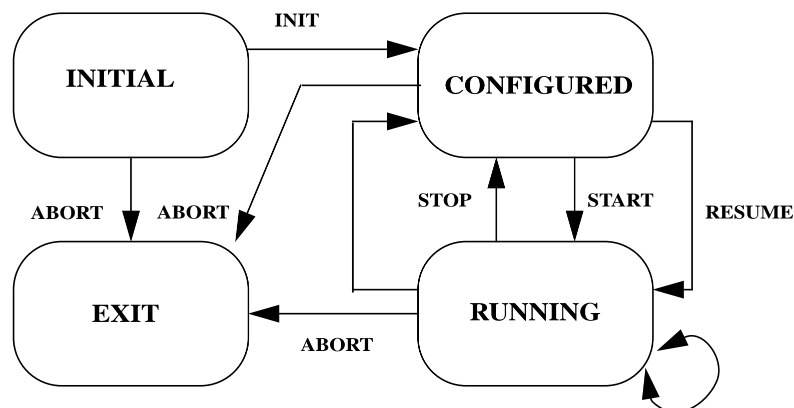

Figure 9: Destination Assignment FSM diagram.

Destination Object

The DFM/FSM object covers the event completion functions (EoT and EoE) and detects the exhaustion of the available resources at the destination (Busy/$\overline{\text{Busy}}$). It is therefore identified by the logical destination identifier (Did). As any other DFM/FSM, object it receives partition commands. Moreover, specific commands related to the event builder high level protocol are treated by the ‘Destination’ objects. The latter are the DESTINATION_GETID command, used to keep track of the number of events being simultaneously treated at the destination, and the DESTINATION_BUSY /DESTINATION_NBUSY command that, when activated, notifies the temporary unavailability/availability of resources at the destination. The EoT and EoE mechanisms are controlled through another set of FSM objects which are associated to the ‘Destination’ object. Each ‘Destination’ object creates, at initialization, a set (=number of events that can be simultaneously processed by the destination) of ‘Event Server’ objects. The set is dynamically activated by the EVT_GETID command which is issued by the ‘Destination’ object, whenever assigned. The partition specific commands accepted by the ‘Destination’ object are the following: DESTINATION_INIT, DESTINATION_STOP, DESTINATION_PAUSE, DESTINATION_RESUME, DESTINATION_ABORT. Figure 10 shows the FSM diagram of the Destination DFM component.

Event Object

This functional component covers the complete functionality required by the EoT and EoE mechanisms. As mentioned in the previous section, the destination object creates at initialization a set of n (=maximum number of concurrent event transmitted to the destination) ‘Event Server’ objects. When a destination is assigned to a new event the related ‘Event Server’ com-

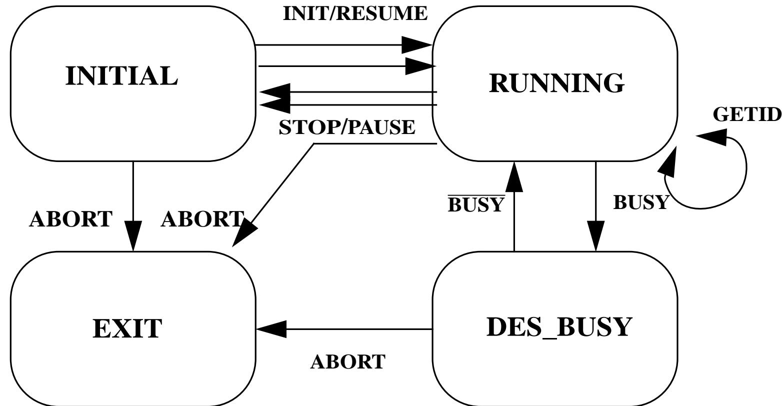

Figure 10: FSM diagram of the destination object.

ponent is activated by the EVT_GETID command which is sent by the correspondent 'Destination' object. The 'Event Server' object is thus dynamically identified by the pairing (Did, Gid). The 'Getid' action starts the counting of the transfers of fragments belonging to that Gid which is achieved by the EVT_EOT command. When the number of transfers is equal to the number of active sources (for the given Gid) the EVT_EOE command is invoked by the object itself and the destination node notified about the completion of the event. At this point the 'Event Server' object is freed and ready to treat another event. Figure 11 depicts the FSM diagram of the DFM 'Event Server' component.

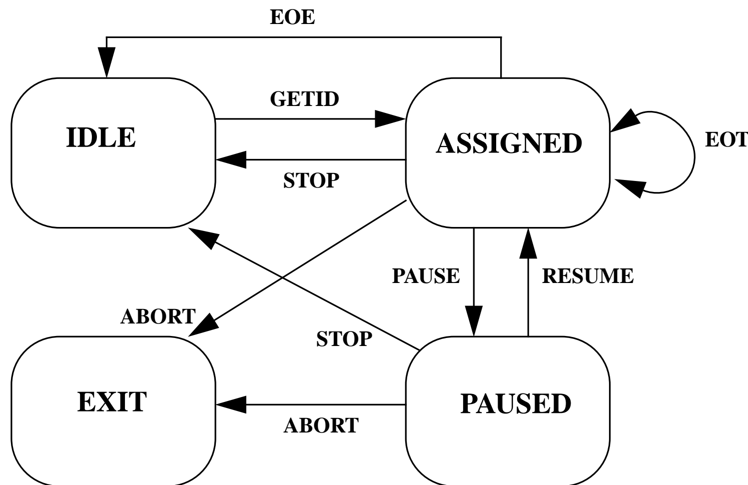

Figure 11: FSM diagram of the Event Server object.

2.3 Sources and Destinations of the Event Builder

2.3.1 Design philosophy

Sources and destinations are tasks of an IOM, the EBIF and SFI respectively: they use all the generic IOM services (scheduler, event management, communication with LDAQ) [4] and follow their design philosophy.

2.3.2 Interfaces

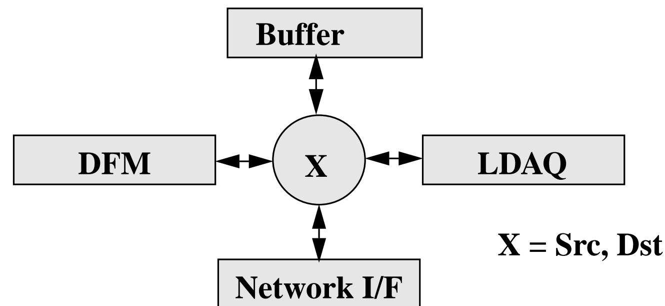

Figure 12: Src/Dst context diagram.

Sources and destinations have common interfaces with a buffer (the input buffer and the output buffer respectively), with the DFM for ensuring the correct flow of event fragments, the LDAQ (via the generic IOM) for run control, monitoring and error reporting purposes, and with the Network I/F which performs the fragment transfer. Furthermore, each of them interfaces via a FIFO mechanism with another task of the same IOM on which they are running. The source is notified by the data collection task of the EBIF when crate fragments are ready to be treated. The destination notifies the output task of the SFI when there are complete events ready to be treated.

Interface with the LDAQ

The communication of the LDAQ with sources and destinations will make use of the services developed for the communication with any IOM. Since for some technologies the network connections have to be opened in two separate steps (first all the receivers and then all senders) the finite state machine developed within the LDAQ to describe the state of a crate allows for a two step configuration phase (preconfigure, configure).

The communication with the LDAQ is necessary for the configuration of sources and destinations, monitoring and error reporting. In particular, besides the parameters which have to be retrieved for the whole EBIF and SFI, the sources and destinations must have access to a parameter list where the translation between logical addresses (Nid = node id) and network specific addresses is made. Each source/destination will establish connections only with the destinations/sources of their partition: the addresses will be achieved via the LDAQ during the preconfiguration phase. The destinations will also need to retrieve the parameters necessary for the $B/\overline{B}$ mechanism and the list of subdetector id’s within their partition.

Interface with the DFM

Sources and destinations will have to exchange messages with the DFM. The source polls on the availability of a message containing the Gid of a fragment and the Did it has to be sent to. The destination has to send an EoT message each time it has received a fragment, it has to notify the DFM whether it is busy or not, and it receives an EoE when a complete event has been (successfully or not) transferred. The details of the API providing these functionalities are described in [5].

The Network Interface

There is a technology independent API providing the calls necessary to setup connections and transfer data through each specific network. This API is described in detail in [6].

2.3.3 Functional decomposition

Source

The source receives a Gid/Did from the DFM. As soon as the corresponding fragment is available, it is sent. With this design, it is necessary to take care of possible desynchronisations between the arrival of fragments and the arrival of Gid/Did pairs. It must be assured that, even in case of an error implying the loss for one event of one of those two elements, the source will be able to go on sending the other fragments. In order to ease the synchronisation between the information coming from the DFM and the availability of fragments in the input buffer, it is possible to develop the source in two independent and time uncorrelated tasks.

When the DFM sends a message with a new Gid and the associated Did, the corresponding fragment is searched in the input buffer. If the fragment is found it is sent, else the Gid/Did pair is added to a list of available Gid/Did pairs.

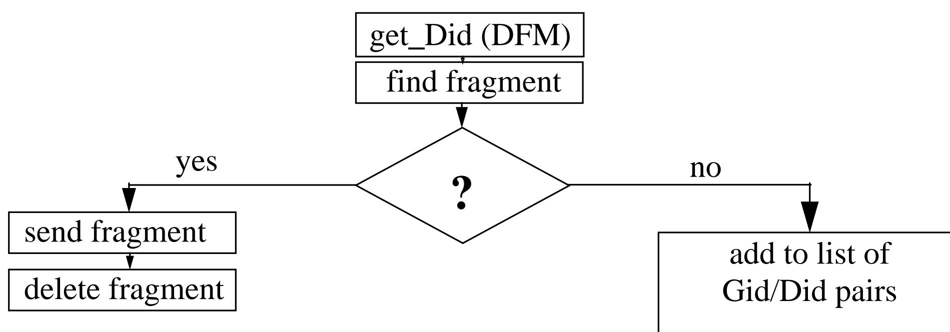

Figure 13: Src module. Logical flow.

```
DFM_get_active() // Poll on DFM messages
DFM_getid(out Gid, out Did) // Implement GetId mechanism
EM_GetById(in Gid, out ptr to data) // Find event in memory buffer
if(EM_GetById(in Gid, out ptr to data) == EM_OK) {
    ebio_send(in Did, in data, in size) // Send event if available
    EM_DeleteByGid(Gid) // and remove from memory buffer
}
else {
    add_to_list(in Gid, in Did) // queue event for later treatment
}
```

The data collection task of the EBIF notifies the source, as soon as a new crate fragment has been built. The source then looks for the destination ID in the locally available list of Gid/Did pairs. If the Gid/Did pair is found, the source reads the event fragment from the input buffer and sends the fragment to the destination via the interconnecting network. Once a fragment has been sent the input buffer is released.

```
is_new_Gid(out Gid) // Poll for a new event
if(find_in_list(in Gid, out Did)== OK) { // ahead for Did
    EM_GetById(in Gid, out ptr to data) // find event
    ebio_send(in Did, in data, in size) // send it
    EM_DeleteByGid(Gid) // and remove from buffer
}
exit
```

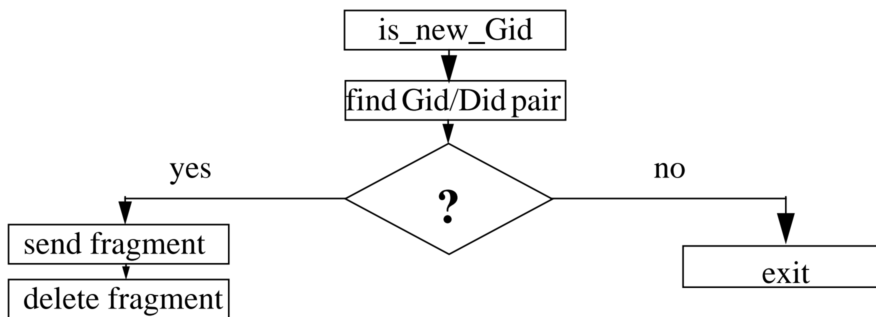

Figure 14: Destination module. Logical flow.

Destination

The destination can be described in terms of three logically separate tasks as shown in Figure 15: It receives fragments, assembles events providing all the necessary headers, and periodically checks how many memory resources are left to the event manager in order to inform the DFM of its $B/\overline{B}$ state.

Task 1 = receiving fragments:)

Figure 16: shows the behaviour of the task when fired by the arrival of a new fragment. When a fragment is ready to be received, the event manager allocates big enough space to store it. The fragment is received directly into the allocated buffer. It is checked whether for that Gid an EndOfEvent has already been received, indicating that the fragment cannot be accepted any more and has to be discarded. Afterwards it is passed to the event assembler (see Section 2.3.4) which will provide the necessary headers and links the fragment to its subdetector header. On successful completion of these operations the End Of Transfer is sent to the DFM.

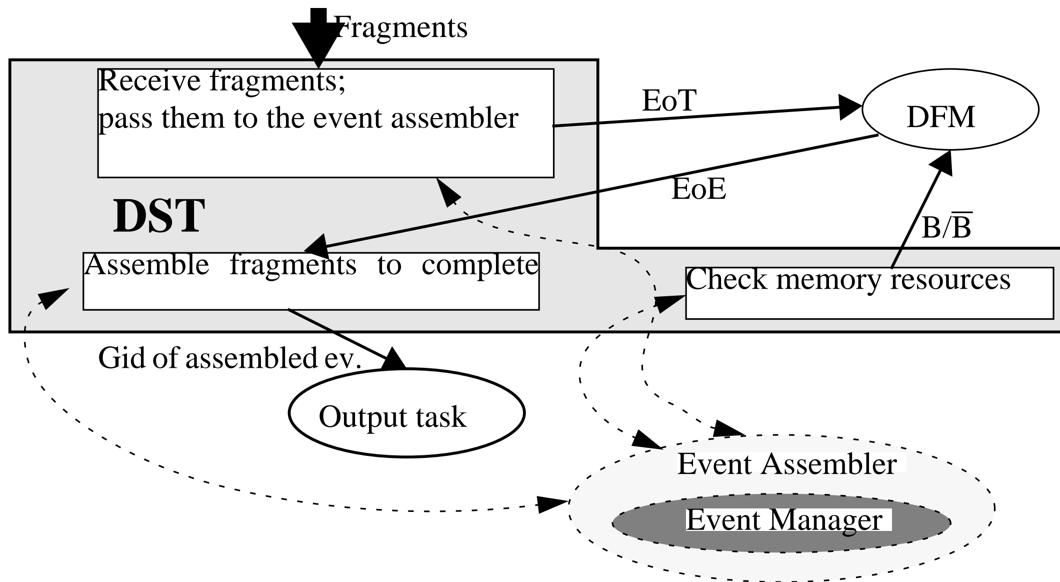

Figure 15: The three different logical tasks of the destination

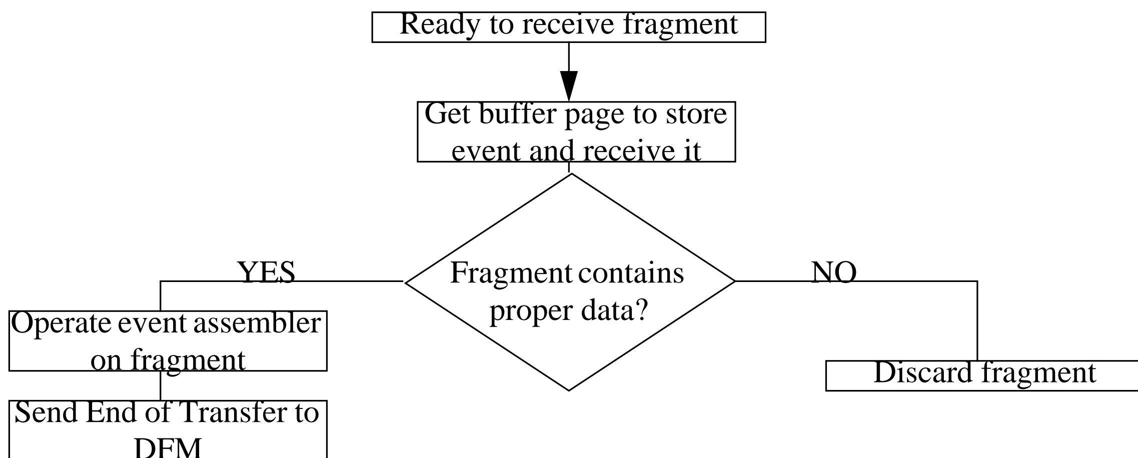

Figure 16: Logical behaviour of the task of the destination triggered by the arrival of a fragment

```
ebio_recv_poll(out yes_no, out Sid) // poll receiver end
EM_GetPage(out PtrFrag)  // allocate space in memory
ebio_recv_read(in Sid, out PtrFrag->data, out size) // read fragment
if(Find_in _list_of_corrupted_events(in Gid) == OK) {
    EM_ReturnPage(in PtrFrag) // discard corrupted event
return
}
BK_AddHeader(in Gid, in PtrFrag->data) // add event header
DFM_EndOfTransfer(in Gid, in Sid) // send EOT if needed
```

Task 2 = receiving an EoE message:

Figure 17: describes the behaviour of the task when fired by the arrival of an EoE message. A function polls on an “End of Event” message from the DFM. The event is assembled and registered in the event manager. If the event is corrupted it is added to the list of corrupted events. Furthermore, the Gid is added to a list containing the ready events (this list will be used by the output task of the SFI).

```
DFM_evctrl_EoE_ready(out yes_no) // poll on arrival of EOE message
DFM_EndOfEvent(out Gid, out status) // read EOE message
BK_Assemble(in Gid, in status) // now assemble the full event
if (status != OK) {
    Add_to_list_of_corrupted_events(in Gid) // if corrupted, take
    notice
}
add_to_ready_events_list(in Gid) // ready to be shipped
```

Task 3 = checking the availability of memory resources:

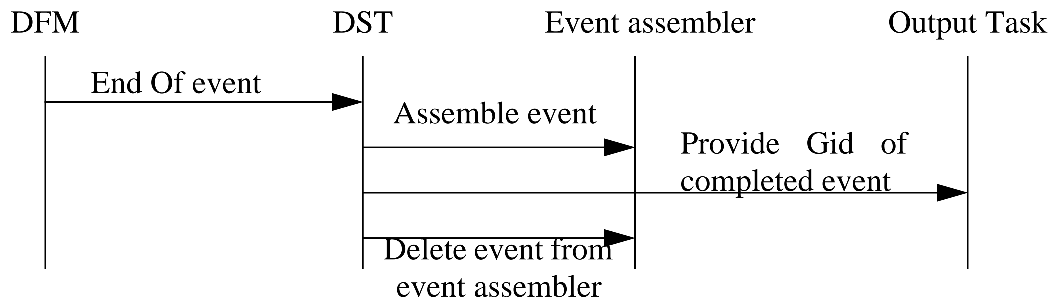

Figure 17: Logical view of the task of the destination waiting on the End of Event message

The availability of resources has to be tested on the basis of the estimation of the number of complete events that can be stored in the event manager. Since several events are being built concurrently and the memory resources are allocated at arrival of each fragment, the availability of many EM pages alone is not sufficient to guarantee the possibility of receiving and building new events. The destination must notify the DFM if it is not capable of being assigned new events using the B/$\overline{B}$ mechanism.

```
BK_GetNevents(out Num_of_Events) // how many events in buffer?
if ((Num_of_Events>= BUSYLIMIT) &&(Dst_is_busy==NO)) {
    set_destination_busy(in YES) // if beyond limit, declare itself busy
    DFM_BusyNotBusy(in Did, in BUSY)
}
elseif((Num_of_EvHeaders<NOBUSYLIMIT)&&(Dst_is_busy==YES)) {
    set_destination_busy(in NO) // if busy and below threshold
    DFM_BusyNotBusy(in Did, NOTBUSY) // tell world no longer busy
}
```

2.3.4 Data handling and formatting

The Event Manager

The event manager is responsible for storing data and providing a means of dealing with them. Its API is specified in [7]. [1]

The Event Assembler

One of the tasks of the destination is the assembly of fragments to complete events containing event and sub-detector headers. In order to perform this activity the usage of the event manager must be integrated with a library which keeps track of the arrived fragments. There is no need to store fragments belonging to the same subdetector in a specific order, but they have to be grouped and preceeded by the corresponding sub-detector header. Furthermore, every event has to start with an event header and, within an event, the different sub-detectors will be ordered. For this purpose an “event assembler” has been developed: it makes use of the event manager for the data and header handling and storage and it registers into a structure the Gid and location of every event and subdetector header.

The event assembler consists mainly of a linked list of arrays and the event manager. At arrival of a new event an array is reserved, for the storage of the location in the event manager of the event header and the subdetector headers, and is linked (with a single linked list) to the arrays containing the same type of information for other events. The linked list mechanism (Figure 18) allows to look for the needed information using the Gid as searching key and to easily remove arrays of completed events.

When the first fragment of an event arrives, the event header and the subdetector header for the subdetector id of that fragment are created, put into event manager pages and their location is stored in an array. Furthermore the fragment is linked, via the EM_LinkPage function to its subdetector header. When the next fragment for the same event arrives, there are two possible scenarios: either it belongs to a different subdetector, in which case the subdetector header has to be created, put in the EM and its location has to be stored in the array, or it belongs to the same subdetector, in which case it is just linked to the subdetector header via the EM_LinkPage function. While fragments for the same event keep arriving, the objects

1. Sources and destinations make an extensive use of the EM making calls to the following functions: EM_Open, EM_GetById, EM_DeleteById, EM_IsFreePage, EM_GetPage, EM_ReturnPage, EM_LinkPage, EM_Create, EM_Reset, EM_Close.

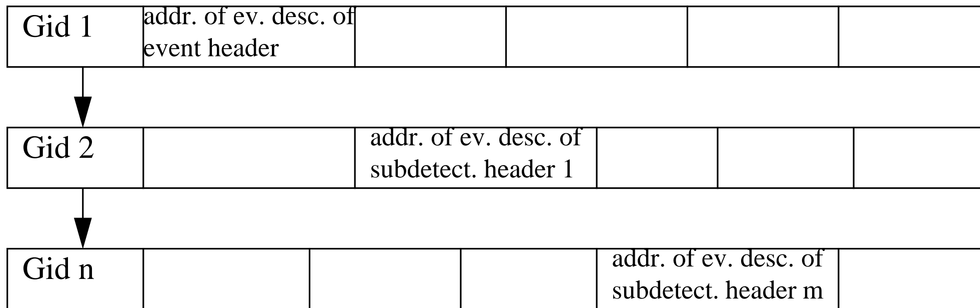

Figure 18: Linked list of arrays, where the location of the event and subdetector headers is stored

depicted in Figure 19: are created in the EM. When the event is completed, it is assembled by linking each subdetector chain to the event header as shown in Figure 20: Using the event manager it is possible at this point to handle the event as an unique object.

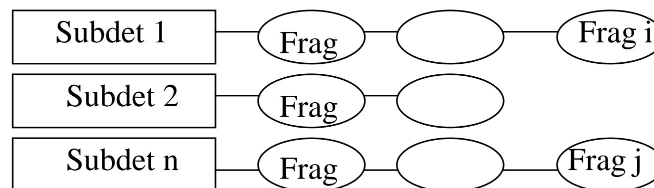

Figure 19: The fragments belonging to a subdetector are all linked with each other in the EM

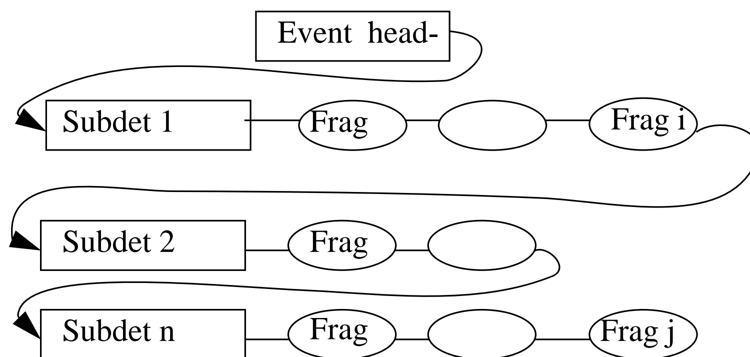

Figure 20: The completely assembled event consists of a chain of linked event manager pages

2.4 Error handling

In this section the possible asynchronous error types that could appear in the EB at runtime are listed. All errors are reported via the same scheme used in all other IOM applications. No error handling policy has been defined yet.

- *Communication errors*: since sources, destinations and the DFM communicate via a network, there is a whole class of possible errors due to the malfunctioning of the technology specific drivers, the PCI/technology interfaces or the network itself. These errors can be furthermore classified in different severity classes. An isolated send or receive error will potentially cause a fragment or a control message to be lost, but does not affect the func-

tioning of the EB itself. Such errors can be classified as WARNINGS. The loss of connection to one ore more destinations as well may cause some fragment losses, but allows the EB itself to continue running. These errors can be defined as RECOVERABLE. On the other hand the loss of connection with a Source requires the EB to be reconfigured and the loss of connection with the DFM blocks the EB high level protocol: these errors have therefore to be considered FATAL for the application.

- *Event manager errors*: all functions related to the activity of event management are subject to errors. Though the lack of memory resources is a severe error and might lead to the loss of event fragments, the back-pressure mechanism implemented within the data acquisition should allow to RECOVER from this error state without crashing the applications.
- *Synchronisation errors*: this error class describes those cases in which one of the EB nodes reacts slower than the rest of the system to the occurrence of an event. As long as the system can recover from it without external intervention in a short time these errors can be considered as WARNINGS and can be handled with internal timeouts or with the Busy/$\overline{\text{Busy}}$ mechanism.

3 Prototype Implementation

The initial implementation of the EB prototype was guided by the decision to use the same networking technology for the exchange of control messages as well as for data transfer. The DAQ-1 approach of making use of conventional drivers to interface with the network implies a non negligible CPU consumption for every I/O operation. It was judged necessary to minimize the number of messages between processors. The functionality of the DFM is, therefore, partially distributed on the DST processors, resulting in a protocol where the EoT message is implemented as a local function call and the EoE mechanism is implemented as a local function call plus a message going to the centralized part of the DFM. The Gid is directly notified by the LVL2 trigger to the DFM and the destination assignment information is broadcast to all the sources within the same partition. For the technologies supporting multicasting and in a symmetric configuration (# of sources = # of destinations), this implementation assures that none of the event builder nodes has to process event data or control messages at a rate dependent on the number of nodes in the system and grants scalability at least at the level of message exchange.

In order to exercise different networking protocols, all technology specific aspects of the EB have been hidden by an API; sources and destinations were made independent from the DFM and high level protocol implementation via another API.

Three technologies with very different approaches to networking where chosen in a first instance to study the Event Builder: ATM, Fast Ethernet and Fiber Channel. ATM is a technology widely used in telecommunications and as backbone for the Internet. It’s main advantage is the standardised offer of Quality of Service techniques which can help in solving contention which arises in the typical EB traffic where all Sources try concurrently to send their fragment to the same Destination. Furthermore it offers a complete end-to-end lightweight protocol (with a software overhead of 20-30 µs per message), AAL5. Ethernet is the most widely distributed networking technology and the capability of effectively using it for event building would strongly reduce the hardware costs of this sub-system. Ethernet has been studied in con-

nection with the TCP/IP protocol which assures data delivery but has the disadvantage of being fairly heavy weighted (with a software overhead larger than 100 μs per message). Fiber Channel was, up to a few years ago, the only technology offering Gbit/s link speed and was therefore considered promising for event building. Its evolution towards becoming the basic technology for disk connections instead of a general purpose networking technology as well as the appearance of Gigabit Ethernet has reduced the interest in further purchasing it. Fiber Channel studies have been documented in (NIM PAPER) and have been interrupted in 1998.

The EB prototype has been implemented in a way which allows its functioning both within the DAQ-1 prototype and in standalone mode. In particular, Src and Dst have been implemented as C-libraries which can be linked into real EBIF and SFI applications or, respectively, into IOMs which generate event fragments when triggered and delete events instead of sending them to the Event Filter farm. In this chapter results concerning the standalone EB will be presented.

3.1 The Trigger Emulation

Different Trigger emulation and event generation schemes have been developed for the EB, depending on the availability of an external trigger element and of a bus (PVIC or VME) which could distribute trigger signals both to the DFM and the Sources:

1. No external Trigger

   The DFM itself generates Gids and assigns them to Destinations as fast as possible. When the Sources receive the GetId message they generate a fragment and send it.

2. External Trigger only to DFM

   The DFM receives Gids at a predefined rate and assigns them to Destinations. When the Sources receive the GetId message they generate a fragment and send it.

3. External Trigger to DFM and Sources

   The DFM and the Sources receive Gids at a predefined rate. While the DFM assigns the Gid to a Destination the Sources generate the event fragment. This implementation is, from a timing point of view, the one which resembles more the operation of the EB within the whole DataFlow prototype in which Readout Crates and DFM operate concurrently and are resynchronized vie the GetId mechanism.

3.2 The ATM Prototype

The ATM Event Builder has been implemented on VME bus based Single Board Computers (SBC), the CES RIO II 8062 (200 MHz), running LynxOS 2.5.1 as operating system. The ATM interface cards as well as the driver are from the company CES, while the switch is a 12 ports switch, the ForeRunner LE 155 from the company Fore.

The Trigger emulation was performed making use of the PVIC or, equivalently, the VME bus to broadcast event identifiers to Sources and to the DFM. The event rate sustained by the EB was calculated every 10 seconds at the DFM and, independently, at the Destinations. This redundant information was used to monitor the balance of the event assignments between Des-

tinations as well as the behaviour of the EB in case that one or more Dsts are halted on purpose.

Figure 21 shows the event rate sustained by the 1 x 1 ATM EB as a function of the number of concurrent events assignable by the DFM to a single Destination. This number corresponds to the number of credits which are given to a Destination within the DFM. When the credits are exhausted, the Dst is considered to be Busy and no new Gids are assigned to it before at least one of the events being built is completed. It is sufficient to have more than four events being concurrently built in the ATM EB to saturate the rate and be in conditions of ideal parallelism for what concerns the operation of the EB applications. This number is very small and points towards the fact that the latency of ATM NICs and switches is low compared to the time which is required to process the events in the applications. The following measurements have been made with four events being assigned concurrently per Dst. This assures to achieve the maximum possible rate, while keeping a good control of the flow of events in the system. If one of the Destinations falls out or sends a Busy message to the DFM, at maximum 4 events, which might have been already assigned, can be lost. Then, no more events will be assigned to that Destination, until a $\overline{\text{Busy}}$ message is issued.

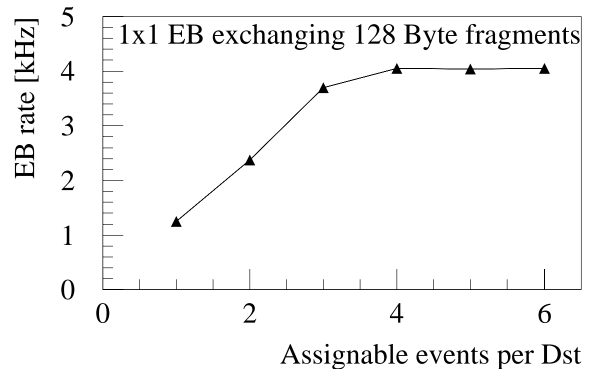

Figure 21: Event rate of the EB as a function of the number of events which can be concurrently assigned to a Destination. Each fragment has a size of 128 Byte.

Figure 22 shows the time requested to build an event from the moment of the Destination assignment to the notification to the DFM node of its completion as a function of the number of Sources involved, for fragments of 256 Byte. This measurement is the most crucial indicator for the scalability of the performance of the Event Builder prototype to the ATLAS EB. A linear behaviour of the latency implies perfect scaling, which means that an N x N Event Builder will have the same performance as a small 2 x 2 prototype. The measurement is very encouraging, although the limited number of nodes available is not sufficient to unequivocally exclude the presence of a week quadratic dependency of the EB time on the number of Source nodes[1]. A quadratic behaviour of the EB time arises if there are one or more functions whose processing time depends linearly on the number of nodes in the system, such as the *atmSelect* function in the DFM or Destinations. Simulation studies on the EB have shown (see Chapter 9) that even a weak quadratic dependency may have dramatic effects on the performance of the ATLAS EB.

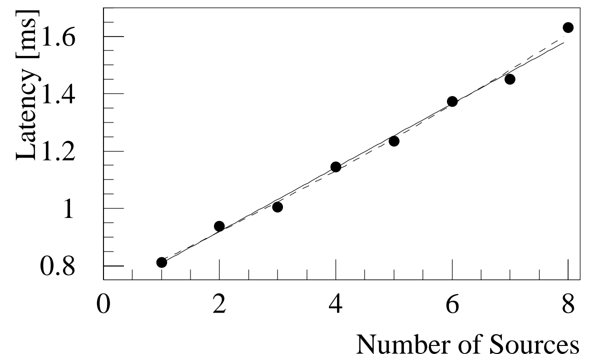

Figure 22: Time requested to build an event from the moment it is assigned to when the EoE message reaches the centralised part of the DFM as a function of the number of Sources involved in the EB. The dashed curve corresponds to a quadratic fit of the data while the continuous line corresponds to a straight line fit[1].

1. Data could be fitted with either a linear function f(x) = 697+111x with a χ^2 of 3.9 for 6 degrees of freedom or with a quadratic function $f(x)=723+93x+2.2x^2$ with a χ^2 of 3.6 for 5 degrees of freedom.

Figure 23 shows the comparison of the event rate achieved by a 2 x 2 Event Builder with and without Data Flow Management. In absence of a DFM the Sources assign the events to the Destinations with a very simple static algorithm based on the event Id, and the Destinations perform the fragment counting and event completion on their own. This comparison is not meant to evaluate the possibility of running the EB without a DFM, which is absolutely necessary for robustness and system's fault tolerance reasons, but only shows how much could be gained in performance in the case that a very fast reflective memory system would be introduced for the exchange of control messages between the DFM and the other EB nodes. The difference is particularly appreciable when small data fragments are exchanged, since in that region the performance is dominated by the software overhead rather than by the network link speed.

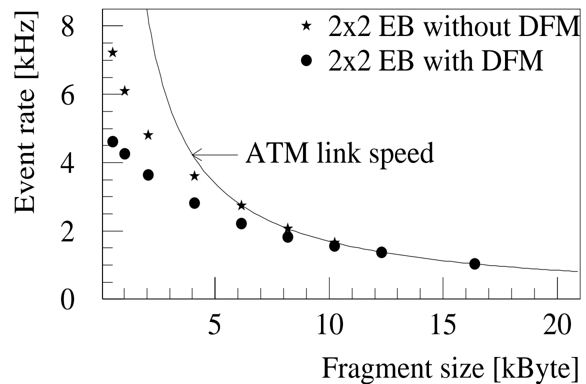

Figure 23: Comparison of the rate achieved by a 2 x 2 EB with and without Data Flow Management

Finally, Figure 24 shows the performance of symmetric EB configurations of up to a 4 x 4 system as a function of the fragment size. The 1 x 1 prototype achieves a lower event rate due to the fact that the Destination has to perform the event assembly for every fragment received, while for the other configurations the rate is limited by the performance of the Sources. The fact that for the three bigger EBs the event rate is the same for big fragments is an indication that the switch behaves as an ideal switch without any problem in serving the ATM full link speed to all the nodes: this is anyhow expected for a small 12 ports switch which is designed to offer an aggregate bandwidth of 2.5 Gbit/s. For small message sizes the EB performance is limited by the software overhead imposed by the EB protocol as well as the CPU consumption due to the fragment ordering and the event formatting operations in the Dsts. As the size of the fragments, and therefore the time to transfer them, increases, the event rate is more and more dominated by the link speed of 155 Mbit/s (~ 135 Mbit/s payload) offered by ATM. For fragments bigger than 8 kByte the EB is completely exploiting the network capability.

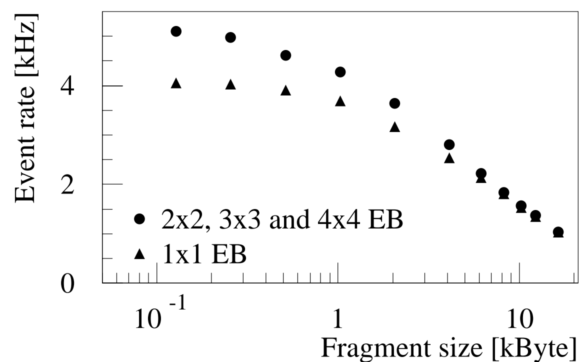

Figure 24: Event rate of the EB as a function of the data fragment's size.

More detailed information on the prototype setup as well as several other measurements are documented in [8].

3.3 The Gigabit Ethernet Prototype

Gigabit Ethernet promises a high transfer speed for the event builder and it is fully compatible with existing Ethernet installations. An extensive program of studies for an Event Builder based on Gigabit Ethernet has been performed, the results are detailed in reference [9].

The setup (Figure 25) consists of 16 PCs running Linux and 2 Gigabit Ethernet switches. Both the switch and the Network Interface Cards (NIC) are from Alteon. The NIC Linux device driver is the one developed at CERN and distributed by Alteon. TCP/IP has been used on top of Gigabit Ethernet to provide end-to-end functionality.

The study has addressed the following issues, when Gigabit Ethernet is used for the event building application: the applicability of the technology, the usefulness of Jumbo frames (as provided by the Alteon hardware), the performance problems due to TCP/IP and the scalability of the setup.To this end the event builder prototype software has been ported to run under Linux on a PC.

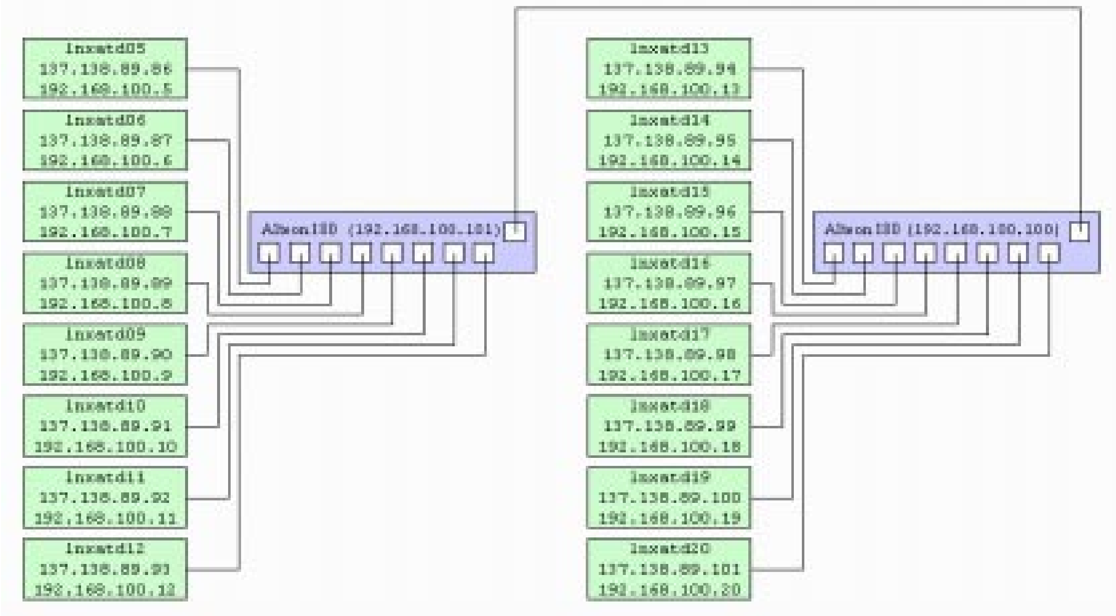

Figure 25: Setup of 16 PCs and 2 switches

In Table 1: we have summarised the values and ranges of the parameters relevant to the performance study.

Table 1: DFM parameters used for this measurement

Event Builder Parameters	# of Sources	1 - 7
	# of Destinations	1 - 7
	Event Builder Scheme	distributed
	# of DFM resources	1 - 50
	Busy	(# of DFM resources) x 1.2
	$\overline{\text{Busy}}$	(# of DFM resources) x 0.2
ebio Parameters	Nagle algorithm	on
	Small message handling	4 Byte control message and message body (up to 249 Byte) sent separately
	TCP buffer size	64kB
Gigabit Ethernet Parameters	MTU	1500

The performance studies are detailed in [9]. Here we summarise the results related to the measurements with the event builder software. First for 1 (source) to N (destination) configurations, then for the N (sources) to 1 (destination) setup and finally for the general NxM case.

3.4 Measurement with 1xN Configuration

In this case, the number of sources was fixed to 1 while that of destinations was varied from 1 to 7.

Figure 26: depicts the EB rate as a function of how many events can be concurrently built at a destination. The rate is linear and independent of the number of destinations up to a "concurrency" value of about 40. Beyond 40 the rate becomes proportional to the number of destinations independently of the value for "concurrency".

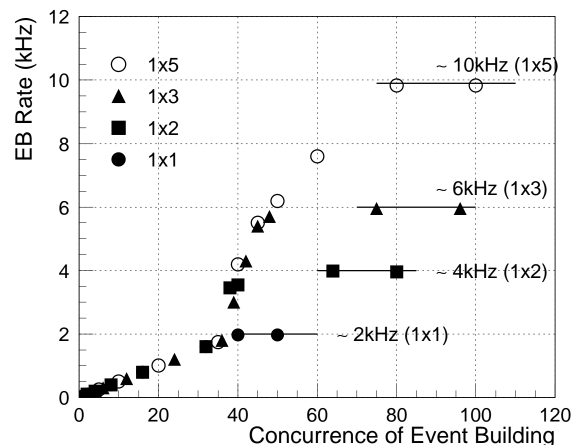

Figure 26: Event builder rate vs. concurrency of event building in 1xN configurations

The event building rate as a function of the fragment size has been measured with a a value for “concurrency” of 50, so as to be in the plateau region for all the sizes with the exception of the 100 Byte fragment (Figure 27). We individuate two areas: one in which the rate is independent of the fragment size (for a given number of destinations) and limited by software overhead (typically TCP/IP) and a second one where performance is limited by the network throughput.

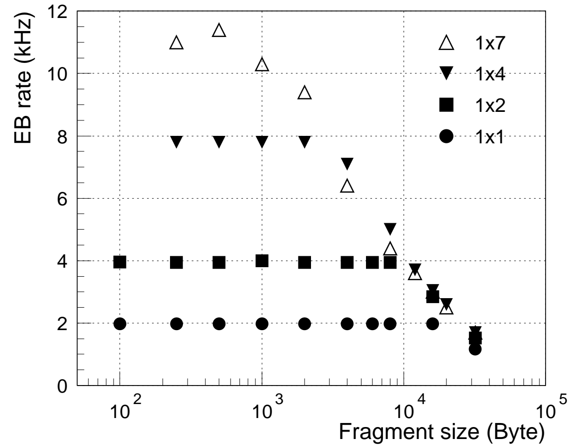

Figure 27: Event builder rate vs. fragment size in 1xN configurations

The bandwidth graph (Figure 28:) is also characterised by two regions: in the first one the performance is dominated by the software overhead of the event builder application and/or TCP/IP (simply called “software overhead”) while in the second region the performance is limited by the Gigabit Ethernet throughput.

For small fragment sizes (less than 20000, 10000, 2000 and 1000 Byte for 1x1, 1x2, 1x4 and 1x7 configurations respectively), the event builder rate is independent of the size and it is proportional to the number of destinations. The proportionality coefficient is about 2kHz, which is determined by the software overhead. This effect has not been understood and was probably an artifact of the system software being used. Indeed more recent measurements have been performed [8] with the standard device driver do not show this 2KHz proportionality factor.

For large fragment sizes, the rate is limited by the maximum throughput per Gigabit Ethernet link on the source side. This is confirmed by the behaviour of the bandwidth with respect to fragment size (Figure 28). The bandwidth reaches a plateau in a region larger than 20 kByte and the total bandwidth does not depend on the number of destinations except for the case of 1x1 configuration. This is due to the limitation of bandwidth of Gigabit Ethernet on source, i.e., the maximum bandwidth of about 50 MByte/s corresponding to 400 Mbps per link at MTU 1500. Which is what is expected, based on the results of the raw measurements (see [9]).

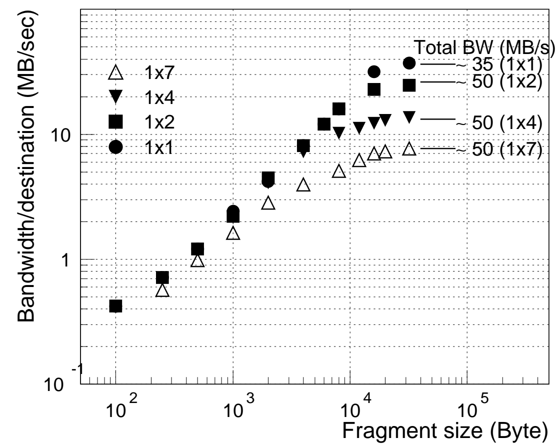

Figure 28: Bandwidth per destination vs. fragment size in 1xN configurations

In the plateau region, the bandwidth of 1x1 configuration is limited to about 35 MByte/s, even if the throughput per link does not reach the limit of 400 Mbps. This suggests that the maximum performance of data transfer of the destination process may be limited to 35 MByte/s, i.e., this is another limitation due to the software overhead.

3.5 Measurement with Nx1 System

For this set of measurements, the number of destinations was fixed to one and that of sources was varied from 1 to 7. Concurrency of event building, TCP buffer size and MTU size were set to 50, 64 kByte and 1500, respectively.

Figure 29: and Figure 30: show the event builder rate and bandwidth per destination versus fragment size for Nx1 configurations.

As for the 1xN case we individuate two regions: the one dominated by the software overhead and that limited by the Gigabit Ethernet link speed.

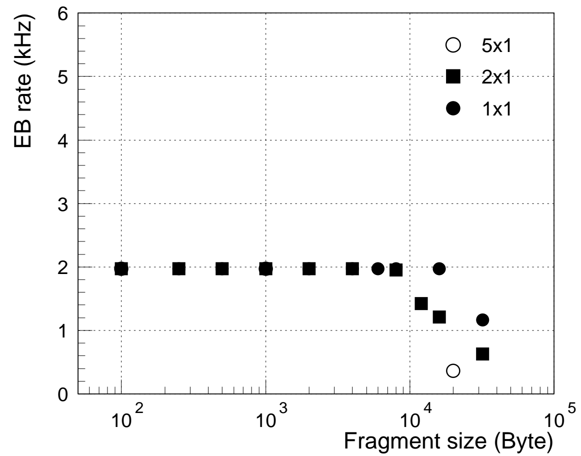

Figure 29: Event builder rate vs. fragment size in Nx1 configurations

The rate is about 2kHz and independent of the number of sources in the region of smaller fragment size (16000, 8000 and 3500 Byte[1] for 1x1, 2x1, 5x1 configurations, respectively). The bottleneck being the destination which processes events at roughly 2kHz in the plateau region, as seen in the 1xN case. Hence the bandwidth is roughly proportional to the number of sources times the 2kHz rate.

For larger fragment sizes, the rate decreases as fragment size increases. The slope depends on the number of sources. The process is limited by the throughput at the destination, as seen in Figure 30:. A plateau of about 35 MByte/s is reached for sizes larger than the transition points.

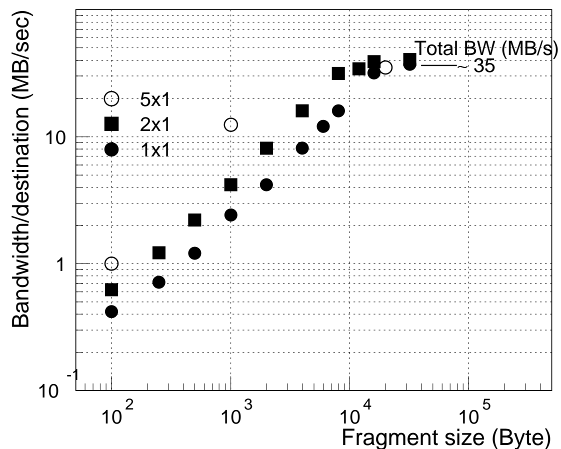

Figure 30: Bandwidth per destination vs. fragment size in Nx1 configurations

3.6 Measurement with NxN Configuration

For this set of measurements the number of sources and destinations is equal and varied from 1 to 7. Concurrency of event building, TCP buffer size and MTU size were fixed to 50, 64 kByte and 1500, respectively.

1. It is difficult to identify where the transition point for 5x1 configuration is, because these results are almost the same as that of 1x1 and 2x1 for the small fragment sizes.

Figure 31: and Figure 32: show the event builder rate and the bandwidth per destination as a function of the fragment size for different event builder sizes.

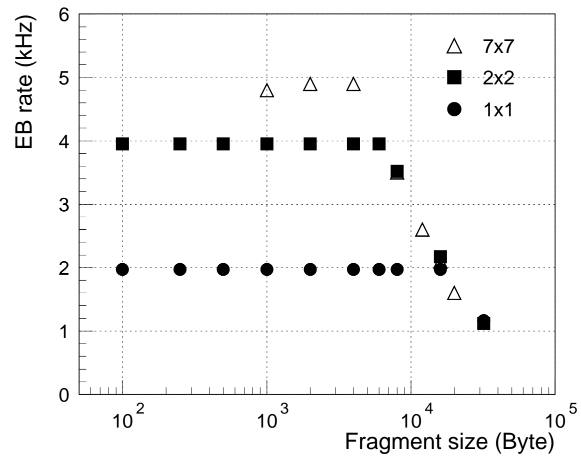

Figure 31: Event builder rate vs. fragment size in NxN configurations.

The behaviour of the rate is similar to the 1xN case while the bandwidth per destination is similar to the Nx1 cases. However, the event builder performance is dictated only by the software overhead.

The rate of 2 kHz x (number of destinations) in the region of small fragment sizes indicates that the rate is dictated by the software overhead as seen in the 1xN configuration. For the 7x7 configuration, the situation seems to be more complicated. The rate of 7x7 configuration reaches only 5 kHz instead of 2 kHz x 7 = 14 kHz. This cannot be explained by the limitation of the software overhead, i.e., 2 kHz and 35 MByte/s. This also means that the event builder system loses scalability for configurations of 3x3 and larger number of sources and destinations.

The event builder rate decreases as fragment size increases in a larger fragment size region, since the bandwidth per destination is limited to ~35 MByte/s by the maximum processing power of the destination as seen in Nx1 configurations.

The limitation on maximum bandwidth per destination of ~35 MByte/s allows total bandwidth of the 1x1, 2x2 and 7x7 configurations to be about 35, 70 and 230 MByte/s, respectively.

In this measurement, the Gigabit Ethernet throughput per link of 400Mbps did not limit the performance of the event builder for the NxN configurations.

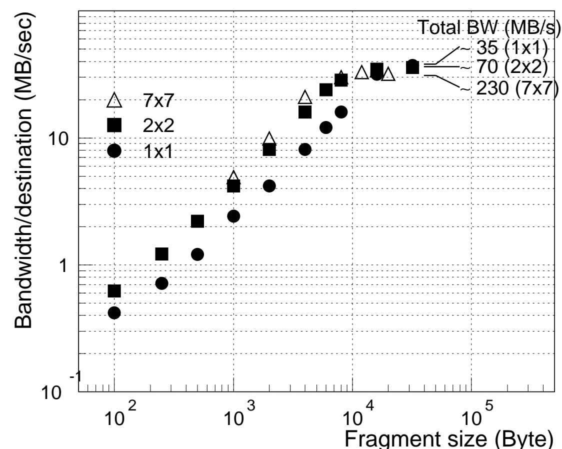

Figure 32: Bandwidth per destination vs. fragment size in NxN configurations

4 Scalability Studies

With scalability studies all those investigations are meant, which are supposed to shed light on the behaviour of the EB as a function of the number of nodes involved in the system. First studies can already be carried out on small scale prototypes, but, in absence of a large scale prototype, reliance has to be put on simulation programs which model the EB. The results of the prototypes have been shown in the previous sections. Here we will concentrate on the modelling and on its results.

4.1 Characterization of the EB Elements

The logical model of the EB is naturally subdivided in three elementary components: the DFM, the SRC and the DST interconnected through a network. On the other hand the performance of the overall EB is dictated by the load of the processor CPUs, by their I/O capabilities and by the links and switch(es) bandwidth. It is, therefore, convenient to factorise the system into network nodes running the applications and a network. It is then possible to further split these elements in terms of queues and resources.

4.1.1 The Network

The network consists of one or more routing elements (resources) and zero or more buffer queues.

4.1.2 The Network Node

The network nodes can be described in a two layer approach. While the applications depend on the type of EB element (SRC, DFM, DST), the structure of the processor with its network interface card (NIC) is invariant The processor is itself a composite of smaller entities: it consists of a resource (the CPU) and three queues (the user space memory, allocated at run time, the kernel receive and send buffer queues). A schematic view of a network node is given in Figure 33. The NIC is again nothing more than a resource for which the input and output data compete.

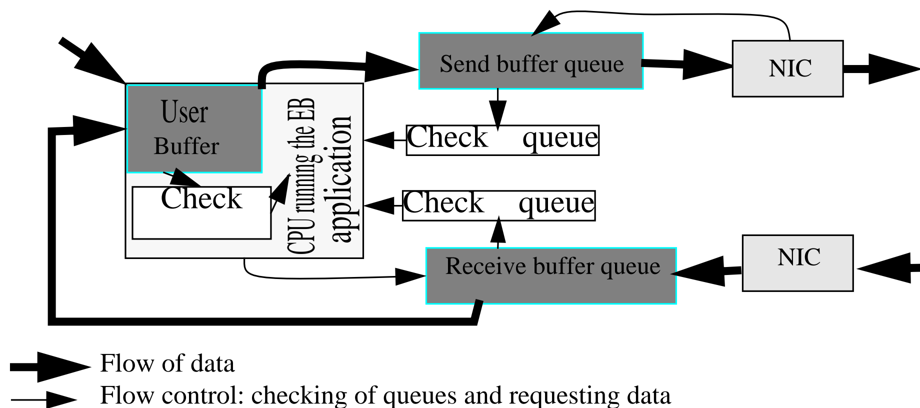

Figure 33: The structure of a network node.
The network interface card is split in two parts in order to allow sender and receiver to make use of different technologies. This takes into account the case in which data and control messages use separated paths.

4.2 The EB model

The model consists of several network nodes running the EB applications interacting with each other over the network as in the prototype. Furthermore, there is a trigger element which distributes L1 ids at a predefined rate. This element is very similar to the trigger element of the prototype and handles back-pressure signals in case that the rate is higher than the one that can be sustained by the event builder. A schematic view of the whole EB model is given in Figure 34.

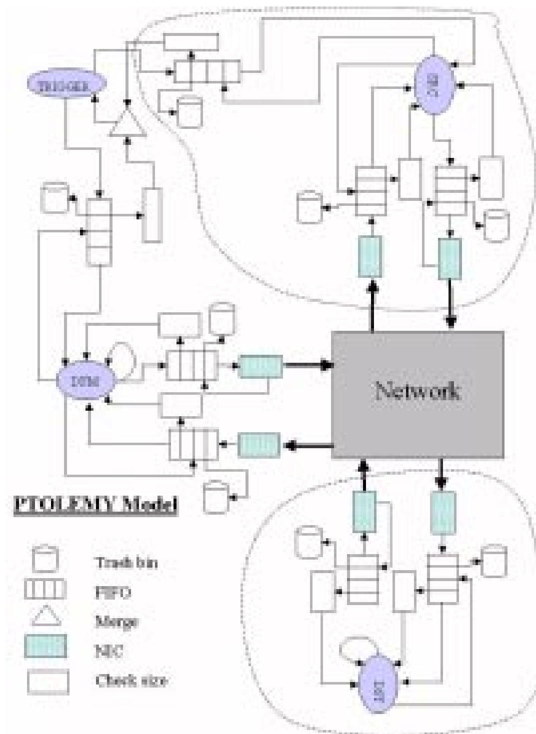

Figure 34: Scheme of the EB model.

The number of source and destination nodes can be configured from run to run, as well as the mean fragment size.

4.3 The Simulation Program

4.3.1 The Simulation Tool: Ptolemy

Ptolemy [10] is a system-level design framework that allows mixing models of computation. The development began in January 1990, under the joint direction of Professors Lee and Messerschmidt at the University of Berkeley.

The ambitious objectives of the Ptolemy project include most aspects of designing signal processing and communication systems, ranging from designing and simulating algorithms to synthesizing hardware and software, parallelizing algorithms, and prototyping real-time systems. In designing digital signal processing and communications systems, often the best available design tools are domain specific. The tools must be able to interact. Ptolemy allows the interaction of diverse models of computation by using the object oriented principles of polymorphism and information hiding. For example, using Ptolemy, a high level dataflow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete event model of a communication network.

In Ptolemy the different computational models are subdivided in domains. For the purpose of event building simulation the only relevant computational model is the discrete event (DE) model. With discrete event simulation (DES) the description of a system in terms of states and changes of state at discrete moments in time is meant. More precisely, DES is a simulation in which the system is modelled in terms of elements which have states. The *state* is a set of variables which fully describe the element at any given time. State changes can only occur at dis-

crete moments in time, at the occurrence of an *event*, and an element cannot change state between two consecutive events. The state of the whole system is given by the superposing of the states of all entities in the system and simulation is carried out by advancing simulation time from one event to the next.

The DE domain in Ptolemy provides a general environment for time oriented simulations of systems such as queuing networks, communication networks and high level models of computer architectures. In this domain an event, the so called particle, corresponds to a change of the system state. The DE scheduler processes events in chronological order. Since the time between events is generally not fixed, each event has an associated time stamp. Time stamps are generated by the block producing the event based on the time stamps of the input event and the latency of the block.

The big advantage of the Ptolemy tool is that it provides, besides the scheduler and the kernel containing the computational model, a series of elements, the so called stars, which are generally useful to any simulation. As an example, the DE domain includes stars describing FIFOs, queues, servers, routers, event generators, logic boxes and delays. Furthermore it provides the hooks for getting graphical information about the behaviour of the applications.

4.3.2 Implementation of the EB model with PTOLEMY

The event builder model has been implemented using the discrete event domain of the Ptolemy simulation tool according to the model described in section 2.1. Several DES elements taken from the palette of stars provided by the tool itself were modified, in order to reproduce correctly the behaviour of the different components of the EB. These elements communicate by exchanging particles. The arrival of a particle carrying a message is the event causing the state transition. The message structure has been defined for the whole simulation to be a six dimensional array of integers (int message[6] = { Gid, size, source id, destination id, status flag, message type }).

For what concerns the switching network, the modularity of the simulation program allows to introduce different levels of detail, depending on the networking technology simulated:

- For ATM, where the traffic congestion avoidance can be handled at the individual nodes via QoS techniques, the switching network can be modelled as an ideal routing element which only introduces a constant delay between input and output.
- For Gigabit Ethernet the switching network with its internal resources should be modelled more carefully except if one introduces some traffic shaping at the application level which avoids that the simultaneous fragments’ transfer congests a single destination link.

In order to be able to easily change timing and bandwidth parameters as well as the configuration of the event builder a C program was written which generates a setup script that can be interpreted by Ptolemy.

4.4 Modelling of the ATM prototype

4.4.1 Simulation Tuning

In order to tune the simulation a set of parameters was extracted from the prototype, such as processing times of the different applications, memory copy speed and overheads for sending

and receiving operations. Complementary measurements to infer bare technology performance parameters such as the switch latency, the link speed and the network interface latency were also performed.

4.4.2 Validation

The model has been validated against the ATM prototype, comparing the event rate for different configurations of the event builder. As shown in Figure 35 the model reproduces the exper-

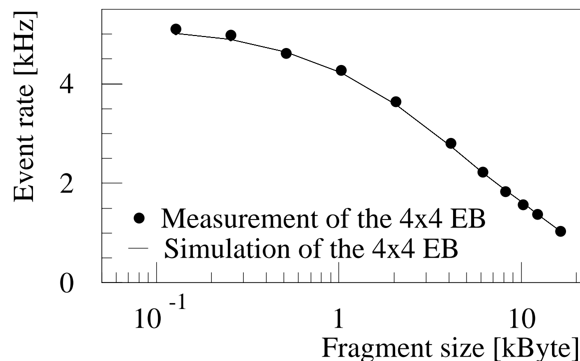

Figure 35: Comparison of the simulation results with the ATM prototype.

imental results within a 5% accuracy. For symmetric configurations the model is event more accurate than that (1-2%).

4.4.3 Extrapolation to a large scale EB

The model was used to infer the event rate which could be achieved in a large scale ATM based event builder. The switching network has been modelled as an ideal element which simply introduces a latency between input and output, without the possibility of data loss, although it should be noted that data loss can occur at the nodes if the kernel buffers overflow.

The latest estimated event size for the ATLAS experiment corresponds to ~2.2 MBytes [11]. The required event building rate is in the order of 1-2 kHz, resulting in an aggregate bandwidth of 4.5 GByte/s. The minimal network configuration, if one imposes that single links shall not be utilized to more than 70% of their capability, has to foresee 400 EBIFs and 400 SFIs for ATM 155 Mbit/s, or, alternatively, 100 EBIFs and 100 SFIs for ATM 622 Mbit/s.

Figure 36 shows the simulation results for a) ATM155 and b) ATM622 using the AAL5 protocol. The extrapolated rate is strongly dependent on the assumptions made on the evolution of processing times as a function of the number of nodes. The rate upper limit is the case in which the processing time per fragment stays constant as the EB system grows [1] while the lower limit is the case which assumes a linear dependency of the processing time per fragment on the number of EBIFs, thus a quadratic increase of the event building time.[2] In the worst case scenario the estimated event rate is not sufficient to cover the required ATLAS performance, and

1. There is a perfect scaling behaviour and the time to build an event increases linearly with the number of fragments it is composed of. The time (in µs) to build an event as a function of the number of sources was measured (with up to 8 Srcs) and fitted with a linear function leading to f(x) = 697+111x with a χ^2 of 3.9

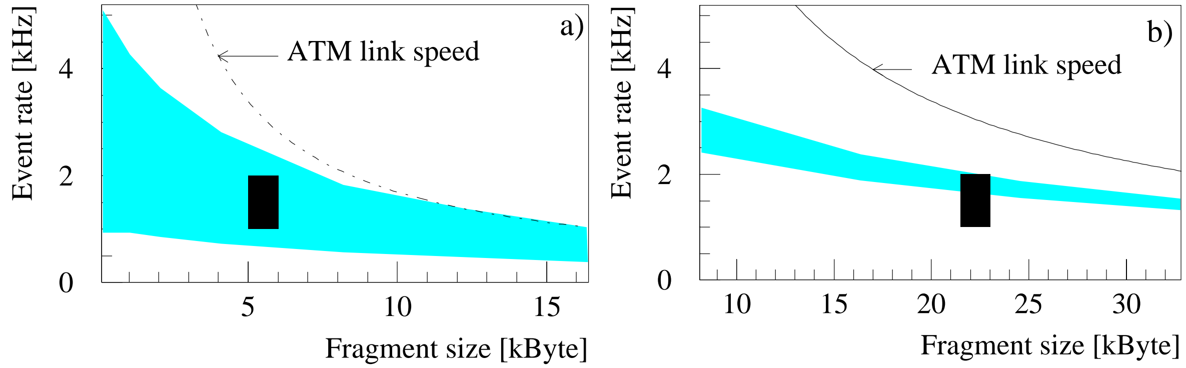

Figure 36: EB Simulation results.
a) 400 EBIFs x 400 SFIs EB with ATM155 and
b)100 EBIFs x 100 SFIs EB with ATM622.
The shaded area shows the expected event rate as a function of the fragment size. The dashed area shows the region interesting for ATLAS assuming that the event data are uniformly distributed over the EBIF nodes.

the results are completely dominated by the processing power of the EB nodes. The time measurements are based on 200 MHz processors; as an example, a 400 by 400 event builder with three times faster processors has been simulated. The results are shown in Figure 37.

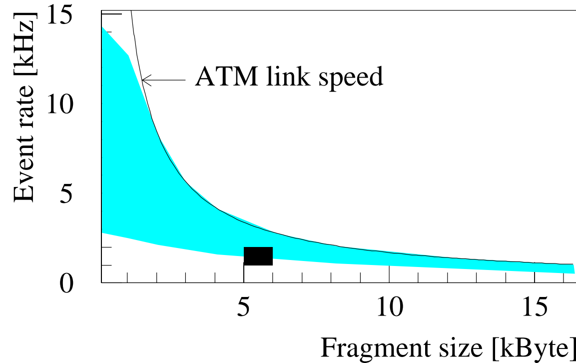

Figure 37: Simulation results of a 400x400 EB with a processor with ~600MHz CPU clock.

4.5 Ideas on the Scalability of the Gigabit Ethernet Event Builder

As already stated, Gigabit Ethernet does not provide QoS techniques which allow to resolve traffic congestion already at the input nodes. On the other hand it is a widely used technology and promises to be the one which will offer the bandwidth required by ATLAS at the lowest price. Within DAQ “-1” no special communication protocols nor optimized drivers have been developed: TCP/IP was used. At the state of the art this implementation presents two critical points:

- the switching network must be capable of handling the burstiness of the EB traffic, providing big enough internal buffers,
- the DFM must send out Gid/Did pairs at a rate equal to $R = event_rate \times NSrcs$.

Both these problems could be solved with appropriate hardware:

2. The time (in µs) to build an event as a function of the number of sources was measured (with up to 8 Srcs) and fitted with a quadratic polynom leading to $f(x)=723+93x+2.2x^2$ with a χ^2 of 3.6

- the network could be oversized in such a way that a burst due to an event could be handled without congestion,
- several DFMs could be introduced in a similar way as it is done within the LVL2 trigger where it is foreseen to have several supervisors. Each DFM would have to send Gid/Did pairs at a rate equal to *R = event_rate x NSrcs / NDFMs*.

Another way of optimizing the EB implementation in Gigabit Ethernet is to introduce traffic shaping at the applications level. In order to compensate for the absence of broadcasting facilities, several Gid/Did pairs could be grouped. The sending rate for the DFM would become *R = event_rate x NSrcs / Grouping*. At arrival of this list of events and their associated destination, the sources could scan and process events starting at a random point of the list. If the grouping factor is chosen to be similar to the number of destinations in the system, in average every source will be sending data to a different destination, hence eliminating the problem of burstiness.

This solution has been introduced in the EB prototype and has shown to be very effective. Results can be found in [8].

5 Fault Tolerance and Robustness

5.1 Fault Tolerance

The EB has been designed to be tolerant against the loss of data fragments. The high level protocol foresees that the DFM can issue an EndOfEvent with a particular status flag, if an event is not completed within a defined amount of time. This mechanism has been implemented in the prototype associating a time stamp to each event, when the first fragment arrives to a Destination, and a timeout mechanism which checks the present time against the event's time stamp and, if needed, sends a timeout signal (POSIX real time timers were used for this purpose).

In principle, the EB design does not foresee the possibility of losing control messages. However, depending on which networking technology is used to transfer them, data loss may occur. In the present prototype implementation these kind of errors are not handled: the loss of a Gid/L1id/Did message to a Source causes the event fragment to stay in the Source buffer until the end of the run. Similarly, the loss of an End of Event message to the centralized DFM node, causes the permanent stay of the event in the DFM structures affecting in this way the Destination assignment policy. Nevertheless some possible mechanism have been introduced in the simulation and have proven to be effective. The loss of control messages can be handled via a timeout mechanism, similar to the one previously described, and additional checks that confirm that the occurrence of the timeout is due to an isolated error and not to the crash of an EB application. As an example, it can be checked that there are messages of the same kind which have arrived for more recent events.

Despite the fact that the Event Builder can be implemented in a way which assures its fault tolerance, frequent errors will, of course, degrade the overall performance. Moreover it has to be verified that the data loss rate in the network is kept at a level which is compatible with the degree of data integrity requested by ATLAS.

5.2 Robustness

The Event Builder has been designed to be robust against the crash of one or more Destinations. The high level protocol includes all the mechanisms for treating this case via the credit based destination assignment scheme and the Busy mechanism. Figure 38 shows the performance degradation as a function of the number of faulty Destinations. The simulation has proven

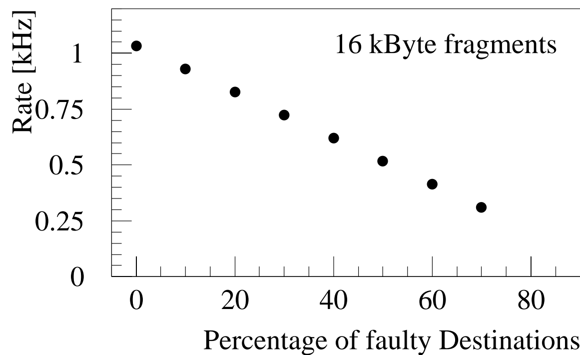

Figure 38: Performance degradation as a function of the number of faulty Destinations.

that the EB performance degrades gradually with the number of faulty Destinations.

The EB cannot, by definition, run without flow management. The DFM is therefore a single point of failure in the EB protocol: its crash causes the whole EB to stop working.

As well for what concerns the Sources, the EB cannot be considered robust. The failure of a Source needs the reconfiguration of the Event Builder in order to change the expected number of fragments to be received per event.

6 Conclusions

The EB elements and protocol have been designed and prototyped. The layered approach taken in the software structuring allows for running the same applications on different technologies.

Two networking technologies, Asynchronous Transfer Mode and Gigabit Ethernet, have been evaluated for event building and prototype implementations have been built for both of them. Both networking technologies have proven to offer the required performance in small scale prototypes.

The Event Builder has been modelled, in order to extrapolate the measurements obtained with the prototypes to the size of the network needed by the ATLAS experiment: the simulation has shown that the actual performance is strongly dependent on the implementation of the Event Builder and in particular on the software which controls the networking operations, but that the event building protocol itself is scalable and that the Event Builder design offers the means to develop this data acquisition subsystem in a robust and fault tolerant way. The evaluation of the performance of bigger prototypes will ease the study of the scalability of this subsystem and favour further investigations towards the optimization of the application software.

The DAQ/EF-1 event builder developments have met the objectives defined at the beginning of the project: an event builder subsystem supporting multiple network technologies has been designed, developed and integrated into the DAQ/EF-1 system. The performance matches the required full event builder ATLAS performance, albeit on a small scale system. The encouraging results from the simulations and the need to address the issues related to system managing and operations call for the continuation of the effort on a larger scale, at least a 32x32 node system. This studies should also be complemented by the evaluation of low latency/low overhead protocols, in particular in the case of gigabit ethernet.

7 References

[1] G. Ambrosini et al., The ATLAS DAQ and event filter prototype '-1' project. CHEP97, Berlin. (1997).

[2] Technical proposal for a general purpose experiment at the large hadron collider at CERN. CERN/LHCC/94-43 (1994).

[3] G. Ambrosini et al., A logical model for event building in DAQ -1, ATLAS internal note, ATL-DAQ-98-112 (1998).

[4] G. Ambrosini et al., The dataflow for the ATLAS DAQ/EF prototype -1. ATLAS internal note. ATL-DAQ-98-095 (1998).

[5] G. Ambrosini et al., The event builder high level protocol API, DAQ/EF-1 Technical note, TN 104 (1998).

[6] G. Ambrosini et al., The Event Builder Network I/O Library, DAQ/EF -1 Technical note, TN 067 (1997).

[7] H.P. Beck et al., The event manager API in ATLAS DAQ/EF prototype -1 DAQ-Unit. DAQ/EF-1 Technical note, TN 071, (1999).

[8] G. Lehmann, Data Acquisition and Event Building Studies for the ATLAS Experiment, PhD Thesis, University of Bern (2000).

[9] Y. Hasegawa et al., The DAQ/EF-1 event builder system on Linux/Gigabit Ethernet. ATLAS internal note, ATL-DAQ-2000-008 (2000).

[10] J.T. Bick et al., Ptolemy, a framework for simulating and prototyping heterogeneous systems. Int. Journal of Computer Simulation, special issue on simulation software development, vol. 4, pp. 155-182 (1994).

[11] P. Clarke et al., Detector and Read-Out Specification, and Buffer-RoI Relations, for Level-2 Studies. ATL-DAQ-99-014 (1999).