UNIVERSITY OF MINING AND METALLURGY

IN KRAKÓW, POLAND

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND ELECTRONICS

Institute of Computer Science

Data distribution system for the DESY/H1 experiment analysis (System dystrybucji danych do analizy dla eksperymentu DESY/H1)

Jacek Nowak

Master of Science Thesis

Computer Science

Supervisor: dr Marian Bubak

Kraków, May 2003

Abstract

The subject of this thesis is the design and implementation of the core system components of a data distribution system for the H1 experiment at DESY, Hamburg. The aim of the project was to develop a framework for online event filtering and reconstruction and offline reprocessing and analysis. The system is running on a farm of networked, Linux based, commodity PC's.

The input data is divided into computationally independent records called 'events', which represent a numerical description of a particle collision registered by the detector. To efficiently use the distributed resources of the PC farm, events are sent from an input node to multiple worker nodes which perform processing of the data. Results are then sent to an output node. The system can be easily scalable thanks to multiple input and output nodes support.

The events are stored in and transferred between repositories which are multithreaded CORBA servers allowing for multiple readers and writers access. It is basically a first-in first-out store, however the repository supports also synchronization of the event flow by implementing barriers. Barriers are special events which must be written by all writers before they can be read, and have to be read by all readers, before they are removed. This allows to distribute data that is important for event processing to all nodes at the same point in the event flow. The barrier algorithm is complicated and was the main design and implementation issue.

We have chosen CORBA as the communication tool, C++ as the main programing language, JTC Threads and pthreads as the threading library.

In the first chapter the background of the system is described, in the second the project goals. In the third chapter two existing systems are discussed and how they fit into the project goals stated earlier. In chapter four a more formal view of the system requirements is presented. Chapter five is the largest chapter and it covers the design of the whole system. UML diagrams are used to present the class structure and algorithms. Chapter six covers the most important issues of the implementation. In chapter seven results of tests are shown. Finally in chapter eight a short summary is presented and future plans are described. Appendix A contains an opinion about this thesis by Alan Campbell. Appendix B the conference papers for Cracow Grid Workshop '02.

Acknowledgments

I would like to thank my supervisor dr Marian Bubak for his valuable help in preparation of this thesis and critical reading. He helped me greatly in putting my ideas to words.

I would especially like to thank Alan Campbell who is the leader of the project at DESY and made it possible for me to participate in the project and to finish it. The barrier algorithm as well as the general system design is the result of long hours he spent on talks and discussions with me. I am grateful for his help in preparation of the first two chapters of the thesis by explaining me how the H1 detector works.

The thanks for valuable advice and help also go to the people with whom I worked on the project over the last two years: Ela Banaś, Marcin Kuta, Janusz Martyniak, Grzegorz Mazur, Tigran Mkrtchyan, Maximilian Vorobiev and Wojtek Zajdel.

Table of Contents

1 Introduction

1.1 HERA and the H1 experiment

H1 is an international collaboration of about 400 scientists from 39 institutes in 12 countries, performing fundamental research in the field of High Energy Physics also known as Elementary Particle Physics. The H1 collaboration has built and operates the H1 detector, one of the big experiments taking data with colliding electron-proton beams at the HERA accelerator.

Fig. 1. DESY terrain

HERA construction has been started in 1984 and data taking begun in 1991. The accelerator is located at DESY "Deutsches Elektronen Synchrotron" in Hamburg (Fig. 1), Germany, and is currently the only electron-proton collider in the world [1],[2].

The main interest of research of the H1 collaboration is to measure the structure of the proton, to study the fundamental interactions between particles, and to search for physics beyond the Standard Model of the elementary particles [3],[4]. This is essentially done by analyzing particle collisions. In order to verify existing theories thousands of collision (events) have to be analyzed. In order to collect and to process such amounts of data, a high throughput computing environment has to be created.

This thesis describes the contribution to the design and implementation of a framework, which is being used at H1 for all computing tasks involving physical event analysis.

1.2 Computing tasks at H1

One of the main computing challenges of high energy physics is the amount of data it has to cope with. Interactions in the detector occur at a rate of 10.4 MHz giving about 100 kB of information each, giving us about one hundred Peta Byte of data a day. Storage of such amounts of data would be very difficult and costly. A method for reducing the data stream before it is written to tape or disk had to be developed. The process of throwing away uninteresting events is called triggering. Triggering is done online and any delays result in data loss. The design of the triggering

system is therefore especially important. Triggering has been divided into four levels. The first three levels are hardware triggers, the fourth level is a software trigger [5].

The data which passes all trigger levels is reconstructed, that means that the signals obtained from the detector are translated into a description of particle trajectories which resulted from the collision. It is than stored on disk or tape and made available to H1 physics for further analysis. The results of such analysis allow a better view into particle physics.

As more experience is gained in data processing bugs and inaccuracies are discovered or new methods developed and added to the reconstruction code. When the improvements become meaningful the H1 group may decide to reconstruct all events collected once more with the new reconstruction program.

Another computing task at H1 is Monte Carlo events processing. Analyzing simulated collision results which are generated using Monte Carlo methods, is important for understanding how the detector works and what influence the reconstruction process has on the results.

1.2.1 Hardware triggering

HERA is a large underground storage ring. Electron and proton bunches are accelerated by oscillating electromagnetic fields to a speed near the speed of light. They are accelerated in two parallel tubes in opposite directions. At two points (interaction regions) inside the ring electrons and protons are brought into collision. These two interaction regions are surrounded by particle detectors (Fig. 2). A bunch crossing happens each 96 nanoseconds. At the collision a lot of energy is released, which results in a so-called 'shower' of new particles. These particles can be registered when they hit the active detector elements. Each detected particle interaction is called a physical event.

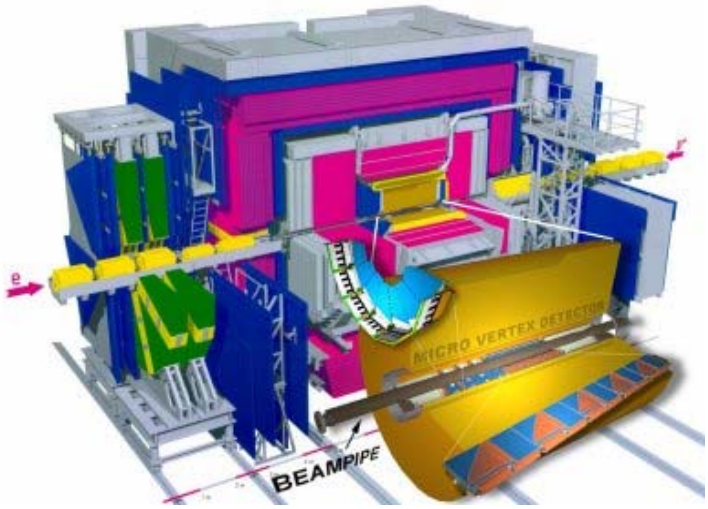


Fig. 2. Particle detector picture

About 10% of the bunch crossings result in a proton-electron collision but nearly at each bunch crossing a collision with a gas particle or the wall of the accelerator occurs also resulting in a particle shower. These events are called background events and in general are uninteresting to physical analysis. The detector can readout events at a rate of about 100 Hz which is not sufficient

to register all events and later decide which of them are interesting. For this reason a facility called 'fast detector' is placed inside the main detector. It can register events at a rate of 100 nanoseconds but has lower resolution than the main detector. However it can identify most of the background events and activate the main detector readout only when an interesting event has occurred. This reduces the rate of events to a level acceptable for the main detector. The fast detector is the Level 1 trigger.

Particles hitting an active part of the detector result in stored analogue electronic signals. When accepted by the Level 2 trigger, special logic based on fast detector signals, these signals are read out, digitized and stored in memory in records called BOS[1] Banks ([6]). A BOS Bank contains information about impulse strength, time, and detector part which registered it.

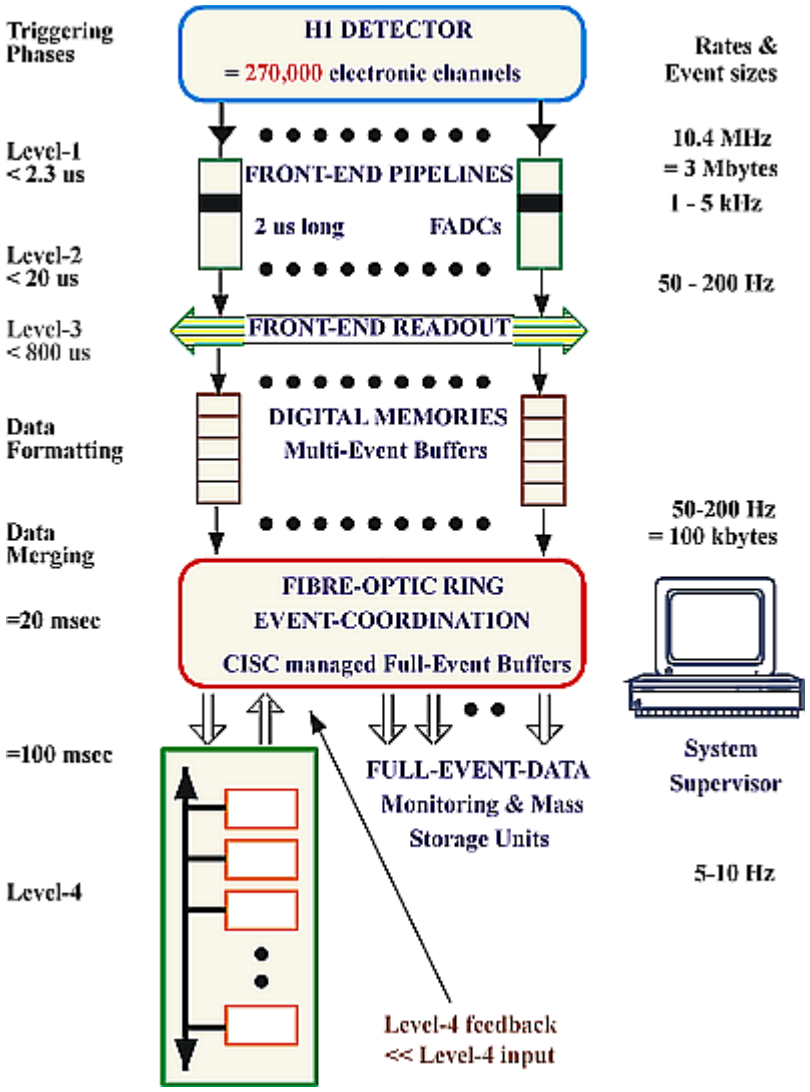


Fig. 3. H1 data acquisition system scheme

BOS Banks from different detector parts arrive at a VMETaxi module ([7],[8]) dedicated for

[1]BOS is a memory management system that enables large and complicated data structures to be realized in FORTRAN

this part of the detector. All Taxi modules are connected in a fiber optic ring. The BOS Banks from all VMETaxi modules are assembled to create a single event. The assembling process is being steered by a Coordination Module (which is also VMETaxi) attached to the ring. At this point the design has foreseen a Level 3 trigger. Its reject decision would stop the readout and free memory in all Taxi modules. However Level 3 trigger has never been actually used.

The scheme of the H1 data acquisition system can be seen in Fig. 3. For further reference about the H1 DAQ (Data Acquisition) system see the H1 Detector paper [4] or the H1 detector web page [9].

1.2.2 The software trigger and online reconstruction

When assembling data from all Taxi modules in the ring is completed we have a data structure containing all information from the detector concerning one physical event. We call this structure a raw data event ([10]). From the Coordination Module raw data events are send via VSB (VME Subsystem Bus) to a process in another VME crate which runs LynxOS (real time Unix [11]). Here the raw data events are put into FPACK format, which is a machine independent data format used widely in high energy physics (see the FPACK manual for more details [12]).

From the LynxOS the FPACK events are sent to a system which performs Level 4 triggering and does the reconstruction. The Level 4 trigger is able to still reject some background events having complete information and more time than the Level 1-3 triggers. The reconstruction process deduces the momentum, the energy and the quantum numbers of the particles which have hit the detector and is sometimes called the Level 5 trigger. This is the last stage of the online process. Now the data is stored on tape or disk and made available for further offline analysis. The data input rate the Level 4 trigger has to cope with is about 6 - 10 MB/s. The corresponding output rate of the reconstruction process is then about 3 MB/s.

1.2.3 Reprocessing

With time, as more and more experimental events are reconstructed and analyzed, some bugs in the reconstruction program, some inaccuracies in calibration constants describing the detectors geometry, may come out and be fixed or new, better reconstruction techniques are developed. When enough such corrections are made or an important problem is discovered and fixed, the H1 members may decided that the changes should be applied to the data that has already been reconstructed. The aim of such reprocessing is to get data with higher quality. Reprocessing

may apply to data from several years and therefore it is a great computing effort. The last reprocessing, which took place in 2002, covered 40 Tera Bytes of data.

The raw data that is needed for reprocessing comes from the output events of the online reconstruction process. The events contain both the input raw data and the reconstructed event.

1.2.4 Monte Carlo simulation

Monte Carlo simulation is done in three steps by three separate applications. The first application is called the Generator. It produces physical events based on the model of physical

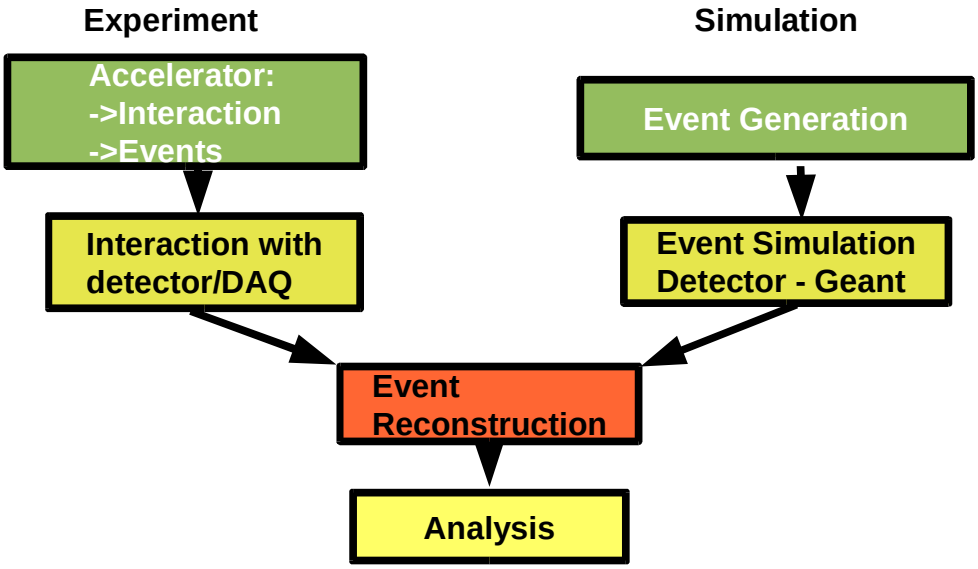


Fig. 4. Experimental data and simulation

interactions. The output is a set of particles described by their momentum, their energy and quantum numbers. Such a virtual interaction is passed to the Simulation application which has a model of all detector parts. It calculates which parts of the detector would be hit, and the signals that would be emitted as the result. The output of the simulation process is similar to the raw data events that we get from the detector. You can see in Fig. 4 how simulation data originates as compared to experimental data.

Hardware triggering is simulated by a software library, and can be performed either at the end of the simulation process or just before the reconstruction. The simulation events are translated to the same format as experimental events and can be put through the same reconstruction process. The resulting events however contain additional data as compared to experimental results. Besides the reconstructed physical event they also contain the source event, which was created by the Generator. By comparing these two events we can see what data got lost and which parameters got

modified during the reconstruction, allowing for better understanding of how the detector works - which physical phenomena can be detected and which will get lost.

1.2.5 Offline analysis

Events in FPACK format which passed the Level 4 trigger and have been put through the reconstruction process contain direct information about the collision in form of particle four vectors. A set of tools written in Fortran has been developed for easy access to this data. Members of H1 and other physicists may analyze thousands of such events and create statistics of processes that happen due to the collision.

1.3 The L45 project

The L45 project has been started with a general detector and hardware upgrade. The old computing system had to be replaced by a new one in order to keep up with the expected increase in data rates. The upgrade touched nearly all parts of the H1 system. The goal of the L45 project was to develop a software framework which could be applied to all main computing tasks at H1 involving event processing. The system should run on the new hardware and take advantage of modern programming techniques and tools.

1.3.1 Upgrade of the triggering system at H1

Before the upgrade the H1 computing system was heterogeneous. A cluster of PowerPC's connected via VME cluster was used for Level 4 triggering and SGI challenge multiprocessor machines were used for reconstruction, Monte Carlo processing and user analysis. The NQS-Batchsystem (Network Queuing System) was used to control job submission and execution. Each time reprocessing had to be done it was decided which machines to use. The L4 triggering and L5 reconstruction were done in separate steps, with data written to mass storage in between.

The upgrade replaced the PowerPC cluster and the multiprocessor machines with a homogeneous farm of commodity PC's and all computing tasks mentioned above were moved to this farm. PBS (Portable Batch System [13]) was adopted as batch system. The L4 triggering and L5 reconstruction were redesigned and merged into one program [14].

1.3.2 Main system tasks and requirements

The main goal of L45 project was to design a framework for running L45 – merged Level 4 and Level 5 trigger – on the new farm of PC's. It was also important that the framework allows for user analysis, reprocessing and Monte Carlo. All the mentioned tasks deal with events as the basic data unit. The design should exploit the computational independence of FPACK events by parallelizing the computations.

The framework should be easily scalable and allow for dynamic resource acquisition and release. It should be able to use legacy Fortran code for event processing and FPACK as the input data format while making the switch to modern object-oriented processing code and data formats fairly easy. New user interface modules should be designed.

1.3.3 A brief history of the project

The L45 project was started in the second half of 2000. My first contribution to the project was in summer 2001 when I came as a summer student to DESY. Since then I have been working on the core system components including basic data structures for keeping events, communication and synchronization between nodes, implementation of different node types, managing startup, stopping, cleanup, crash recovery of nodes, implementation of functions which allow control of the system, design of the Controller module.

The deadline for the framework was the end of 2001 and by this time the first functional version of the system was ready. The time until summer 2002 was a time of consolidation and feature upgrades. The system is currently fully functional. L45 reconstruction and user analysis tests have been successful. Still code cleanup and minor feature enhancements are performed occasionally.

2 Goals of the thesis

2.1 One framework for all processing tasks

The H1 experiment team is performing four main computing tasks - online filtering and reconstruction, reprocessing, offline reconstruction, and Monte Carlo analysis. L45 project goal is to design a framework which could be applied to all these processing tasks.

The data stream coming from the experiment is divided into runs. Each run is associated with a set of calibration constants. They contain information about detector settings which can be changed to better fit the conditions that are in during data taking. Run stop, run start and calibration constants information is inserted into the data flow just like ordinary events. The difference is that they change the way events are being processed. It is necessary that run stop, run start and calibration constants reach each process before data from the next run arrives there.

Online triggering has real time requirements. Data is streaming from the experiment as successive collisions are detected. If the system can not handle the data stream, information is lost. Currently the input data stream is about 7 MB/s and the output data stream about 3 MB/s, but we want the system to be scalable to handle also higher data rates.

The library which performs L45 reconstruction and triggering is carefully developed as it is used by the whole H1 experiment. Most bugs are found before it is applied to real experimental data. Still we have to consider a situation when a program crashes or deadlocks. Such situation should not cause the whole framework to stop, a quick recovery must be possible to eliminate or at least lessen the data loss.

The input data is received from a VME crate running LynxOS via a socket.

A special logging program divides and checks the output data stream. The result of such checks are updates to the calibration constants. These updates should be inserted into the input data stream as soon as possible.

Monte Carlo analysis and reprocessing are very similar to online triggering. The main difference is that the input data comes from a file which has been generated earlier and not from the detector. Hence there are no real time requirements and only the throughput is meaningful. The reconstruction code is here also well tested and common for H1. Fast recovery is here not the highest priority but rather the ability to tell how far the processing got before it crashed. This can not be deduced simply by checking event numbers in the output file since processing on multiple machines in parallel might have changed their order.

After the data from the experiment has been filtered and basic reconstruction has been performed, the events are stored on disk or tape and made available for the H1 group members for analysis. Each member can write his own analysis program and run it on some of the experimental data. Requests from users to perform their computation are queued by PBS batch system, and should be fulfilled as soon as some processing power is free for use.

Because user analysis requires user programs, a simple way to link user libraries into the system is required. Also user programs are not so well tested as online or Monte Carlo reconstruction code. Therefore user analysis should be run in a special safe mode to decrease the possibility of crashes and data loss.

2.2 Utilization of the PC farm

A farm of commodity dual-processor PC's has been adopted as the base hardware for the new computing environment at H1. The PC's are running Linux and are connected by internet. In order to fully utilize the processing power of the PC farm we have to exploit the parallelism of the reconstruction process, while being able to dynamically acquire and release resources of the PC farm as needed.

Each FPACK event is a computationally independent set of data. The easiest way to exploit the natural parallelism of the problem is to distribute the processing of single events. Each event is sent to only one node for processing, but, as mentioned above, some data, like the calibration constants or the markers, has to be distributed to all worker nodes. We also want that groups of events separated by calibration constants or markers don't get mixed up.

The PC farm is large enough for more than one computing task to be performed there at the same time. For online reconstruction it is important that more PC's can join the processing if the throughput is not sufficient to handle the input stream. In general we want the system to be able to free or acquire resources without any interruptions to the data flow.

The operations performed by a process can be roughly divided into I/O and calculations. The calculations require processor time while network I/O is handled mainly by the network interface card. The system should be able to do both operations in parallel to optimize PC utilization.

The PC are dual processor machines. The reconstruction program is written in Fortran and is single threaded. Running two reconstructions in two separate threads or processes should be possible.

2.3 Using modern programming techniques

The use of low level programming languages has been the cause of high maintenance costs and low flexibility in the past. The system should take advantage of new programming techniques and languages, which have been developed over the last years, to reduce the development time, increase scalability and flexibility of the system. This includes choosing a modern object oriented programming language with an interface to a high level, standardized communication library. The programming language should allow for easy access to Fortran routines. Increasing performance in places in code where it is especially needed should be also possible. The communication library should neither require developing a communication protocol nor designing of special server facilities inside the system. The library should have bindings in many languages so that developers of additional modules are not limited to the programming language that has been chosen for the main system components.

2.4 Objectives of the thesis

High throughput computing has always been a requirement in HEP (High Energy Physics) as it has to handle data amounts seldom found in other fields of science. With the construction of particle accelerators and detectors constantly becoming better, creating a system which could handle the requirements is a challenge, event with increasingly faster computer architectures. All HEP experiments have similarities but also specific problems and needs which the computing system has to fulfill.

This thesis describes the development of a data distribution and analysis system for the H1 experiment at DESY. It addresses all its specific computing requirements but at the same time, by applying modern, object oriented design and programming techniques, an attempt is made to create a framework which could be easily adopted to other similar problems.

3 Similar systems

3.1 Condor

Condor ([15]) is a specialized batch system working in distributed UNIX/Linux/ WindowsNT environment. It is designed to manage compute-intensive jobs with high throughput requirements. It has a queuing mechanism, a scheduling policy, priority scheme and resource classification mechanisms. It is especially designed for using non-dedicated resources, that is resources not owned or managed by a centralized resource manager.

3.1.1 Goals

The general idea behind Condor is that with PC's standing on nearly each desktop and often standing idle, we have huge resources of unused computing power. Condor is an attempt to retrieve this processing power. Owners of PC resources would join voluntarily a system in which they could offer their free processing power to other members of the system. In exchange the idle processing power in the system would be made available to them if they need it. Condor is responsible for managing user request and finding idle computers in the system.

The system must be especially sensitive to user requirements. People will only share processing power of their PC if they can be sure they can retrieve it whenever they need it. A resource description language is necessary so that users can precisely describe how and when their machine can be used. A similar job description language is necessary so that users can describe requirements for their computations.

3.1.2 Design

On each machine which decides to join Condor a background process has to be started, which waits for the machine to become idle. On some other machine the central manager is running. It keeps a job queue and when a machine which matches the job description becomes free, it starts the process there. Its responsibility is also to move processes away from machines which stopped being idle. A group of machines with one central manager is called a pool. Pools may be hooked together in a hierarchical manner.

A checkpoint mechanism has been developed to allow for moving processes from one

machine to another. A checkpoint is a snapshot of the process made periodically, containing all information about the process needed to restart it from that point. Checkpoints are also used for crash recovery.

All system calls made by the program are intercepted by Condor and if necessary redirected to other machines. Hence the process is totally unaware of its location.

In order to take advantage of system call redirection and checkpointing the program has to be linked with Condor libraries. There are some limitations for programs which want to use checkpointing, like no multithreading or interprocess communication allowed (refer to Condor manual for details). However linking with Condor libraries is not required for a program which is to be run in a Condor pool.

Each job can be run in one of the four Condor universes:

1. standard universe - checkpoints and system calls redirection for programs linked with Condor.
2. vanilla universe - no checkpoint or system calls redirection.
3. PVM - ability to dynamically create PVM machines.
4. Globus - job submission using RSL strings.

3.1.3 Characteristics

- Specialized batch system for use of idle resources.
- No changes to code necessary.
- Checkpointing and system calls redirection.
- Jobs submitted to Condor can be run on GRID.
- Job and resource description language.

3.2 DIANE

DIANE - Distributed Analysis Environment ([16]) has been developed by Jakub T. Mościcki at CERN. It is a CERNA IT/API project to study the requirements and prototype a middleware distributed environment for parallel data analysis for LHC. It targets mainly parallel ntuple analysis and parallel detector simulation, however it is not limited to any specific application. It was also an attempt to see how API products can be integrated with GRID.

3.2.1 Design

DIANE is component based framework for parallel cluster computation. It is application-oriented featuring a master-worker (see Fig. 5) model which is common in HEP applications It is application independent because applications can be loaded dynamically in plugin style. The

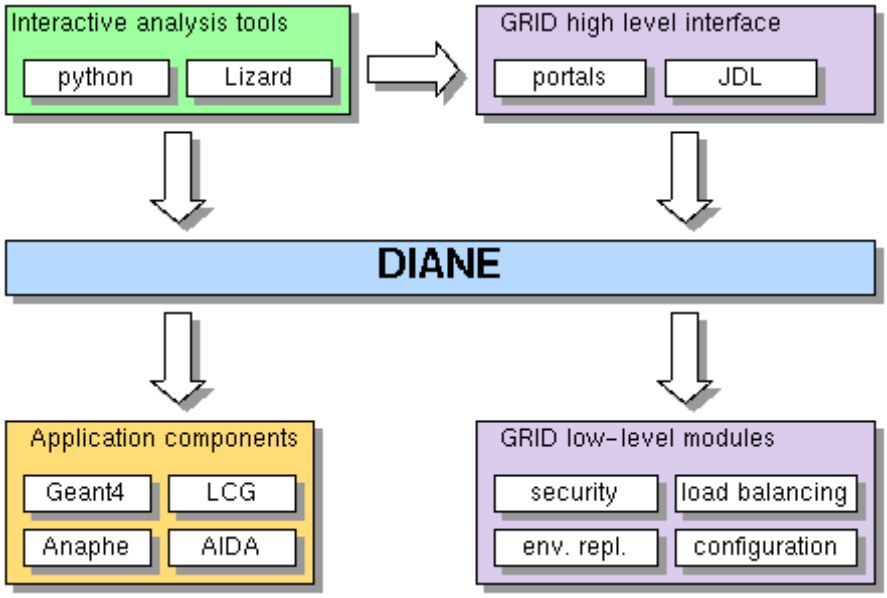


Fig. 5 Picture from Jakub Moscicki presentation DIANE CERN seminar 11.09.2002

framework communicates with applications through a well defined abstract interface. This allows to shield user applications from changes in 3rd party components as well as from changes in DIANE's architecture.

It is designed as an integration layer between applications and the GRID (see Fig. 6). It implements high level GRID interface, allowing to treat the system as a GRID computing element

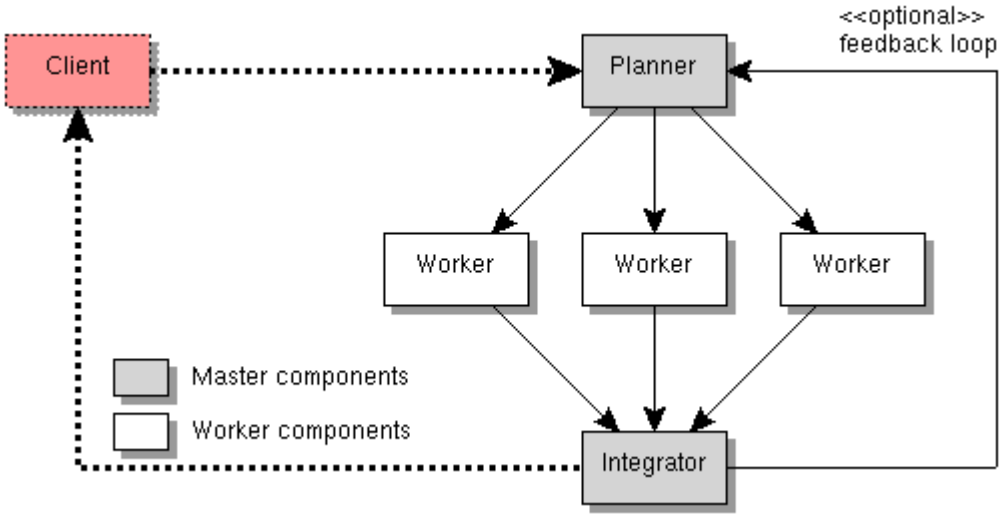


Fig. 6. Picture from Jakubk Moscicki presentation DIANE CERN seminar 11.09.2002

but also uses low-level GRID modules for security, load balancing, environment replication and similar. The system is protected from the instability of GRID technology by packaging modules into pluggable components.

The core of the system is based on CORBA and CCM (CORBA Component Model).

3.2.2 Main characteristics

- Designed to run on a cluster.
- Acts as a semi interactive batch system.
- Highly modular and component oriented.
- Master worker parallel processing model.
- No low level load balancing - static data partitioning.
- Using low level GRID modules. May be accessed via GRID enabled web portals.

3.3 Summary

Most of the freely available tools for distributed, scientific computing focus on job/resource managing making them specialized batch systems. The L45 project targets at creating a more low level system. We hope to be able to address more of the requirements H1 computing has by exploiting the inner parallelism of the analysis which is event-based. The system does not address directly resource managing aspects, offering only an interface which could be used by a higher level batch system.

4 Requirements

4.1 Utilization of the PC farm

The system will run on a farm of dual processor PC's. It must be able to dynamically allocate and free farm resources. The administrator should be able to do it by removing a machine from the system or adding one without having to stop the system or loosing data. The processes should be multithreaded so that disk I/O and network interfaces can be operated in parallel and the dual processor architecture is fully used.

Req.1 A PC assigned to the system at startup can be removed at runtime without any data loss or having to stop the system (Fig. 7).

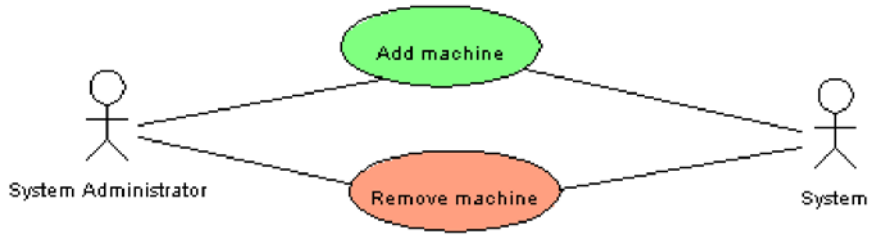


Fig. 7 Add/remove machine use case diagram

Req.2 A PC which was not assigned to the system at startup can be added at runtime without having to stop the system (Fig. 7).

Req.3 Each process should be multithreaded so that disk I/O, network operations and computation can be done in parallel.

Req.4 On multiprocessor machines computation should be done in more than one process or thread.

4.1.1 Throughput and scalability

The data input rate the system has to cope with when doing online triggering is about 10 MB/s. The corresponding output rate is about 3 MB/s. This are the minimal requirements for the system. As these parameters may change in the future we want the system to be scalable to cope also with higher input and output rates. The input and output data streams may be split, so multiple input and output points to the system are possible.

Req.5 The system should cope with an input rate of 10 MB/s and an output rate of 3 MB/s.

Req.6 Higher input and output rates should be possible by supporting multiple input and output points if necessary.

4.2 Data handling

4.2.1 Data format

An event describes a collision registered by the detector. The numerical values are kept in FPACK format. Data in FPACK format is divided into records. This records can be either physical or logical. Physical records are constant sized (the size is usually adjusted to get best I/O performance on the disks), and can therefore contain a part of an event, a full event, or even multiple events. Logical records contain always one event and their size varies.

Req.7 The system may receive logical or physical records as input data and must be able to deal with both. Information which allows to distinguish logical and physical records is contained in the record header.

The format of an FPACK record header is described in [12]. Logical records will be further referred as “FPACK Events”

4.2.2 Data input

The input data may come from two sources either from a file or from a socket.

Req.8 The system must be able to read data either from a file or from a socket.

4.2.2.1 Reading from a file

Reading from a file is done when offline analysis is performed. An FPACK card file defines the file or files data should be read from and the reading mode. A Fortran routine exists for parsing card files line by line. It is declared:

```
void fparm(const char* cmd, long len)
```

where 'cmd' is a line of the card file and 'len' its length. The subroutine which actually reads the data can output it either to a file or it can call a function, which, in C namespace, is declared as follows:

```
void fwspec(int *LUN, int *NWORDS, int *BUFFER, int *IEND)
```

where IEND is the error number, BUFFER is a pointer to the data and NWORDS the length of BUFFER. An implementation of *fwspec* should make a copy of data in BUFFER as ownership of it

is kept by the Fortran routine which calls *fwspec*. Also IEND should be set to 0 if no error occurred. When end of file is reached or an error occurs while reading a function declared as:

```
void fcspec(int *LUN)
```

is called. For further reference to *fparm, frspec, fwspec, fcspec* refer to the FPACK manual [12].

Req.9 Given the FPACK card file location the system should be able to open and parse it using the `fparm` function (or a compatible one). The data should be received by implementing the `fwspec` function (or in a compatible way). Reading should stop when `fcspec` is called.

Req.10 If the given FPACK card file can not be found the system should stop immediately writing to standard output the cause of failure.

4.2.2.2 Reading from a socket

Reading from a socket is required by online triggering. The raw data is packed into FPACK records on a remote machine and made available via a socket.

Req.11 Given the socket address the system should be able to read subsequent FPACK records until end of file is reached or an error occurs.

Req.12 If a connection to the given socket address can not be established the system should stop immediately writing to standard output the cause of failure.

4.2.3 Data output

The system output consists of FPACK Events containing results of the processing. The data is written to a file or multiple files. This is done by a Fortran routine which acquires subsequent records for writing by calling a function:

```
void frspec(int *LUN, int *NSKIP, int *IRC, int *NTOT,
            int *BUFFER, int *IEND)
```

where BUFFER points to a memory address where the data should be copied, NTOT is the size of BUFFER and IEND the error number which should be set to 0 if no error occurred.

Req.13 The results should be put out by implementing the `frspec` function.

4.2.4 Data processing

The processing of events (reconstruction, software triggering) is also done by Fortran routines. This routines accept FPACK Events as input data and put out results also as FPACK

Events. This routine uses *fwspec* to output results and *frspec* to get data.

Req.14 Feeding data into the processing routine and collecting results should be done using `fwspec` *and* `frspec` *functions.*

4.2.5 Grouping of events

The input data consists of events. The order of events is, in general, not important so that events may be put out by the system in different order then they were at the input side. We want however to be able to group events. The order of events is still unimportant but events from different groups should never mix with each other.

Req.15 Events can be grouped in such a way that events belonging to one group at the input side are still in the same group when they are put out by the system.

4.2.5.1 Run start, stop events

At the end of each run a run stop event is inserted into the data flow ending the last run, and a run start event, starting the new run. Sometimes run stop/start events of empty runs are not inserted into the data flow. We expect the system to discover such situation at the input side and create emergency run stop/start events.

Req.16 Events from different runs should be divided by a run stop and run start event.

Req.17 If a run stop/start event is missing in the data flow it should be created by the system.

4.2.5.2 Additional grouping of events

We want also to be able to add additional grouping of events inside runs.

Req.18 The system should support additional grouping of events inside runs.

4.2.5.3 Distributions of calibration constants

Calibration constants are special events which interpreted by the processing code change the way events are being processed. Calibration constants come always after a run start event of a new run. Additional update calibration constant events may be inserted also in the middle of a run.

It is important that calibration constants reach each processing program before the events they apply to. This means that calibration constants are also grouping events.

Req.19 The system should support grouping of events by inserting calibration constants and 'update calibration constants' into the data flow.

4.3 Fault tolerance

4.3.1 Errors in input data

Input records contain sometimes errors which can be easily found by validating the record headers. Such errors often make it impossible to extract events from this records. If this is the case a record should be thrown away by the system.

Req.20 If errors in the input records are encountered which make it impossible to extract events from them, these records should be removed from the system and a message describing the error type should be written to standard output (Fig. 8).

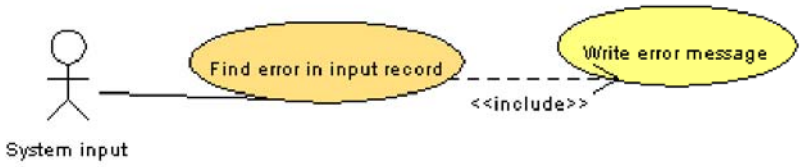


Fig. 8. Error in input data use case diagram

4.3.2 Event processing

The processing programs sometimes contain bugs which may cause them to crash or deadlock on certain events. The system must be able to continue processing in such a case. If possible the event which caused this situation should be identified and left for further investigation.

Req.21 The system should be able to realize about crashed processes. The necessary cleanup should be performed and the process restarted. Whenever possible the event which caused the crash should be saved (see Fig. 9).

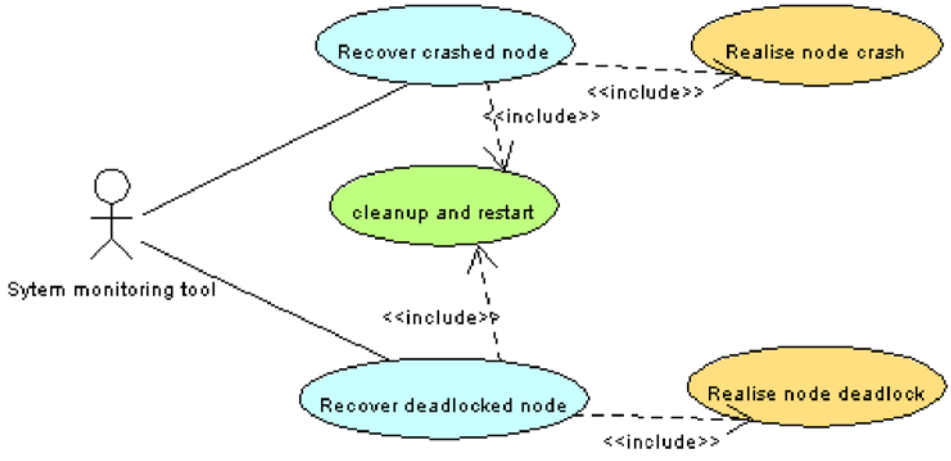


Fig. 9. Recover from crashes use case diagram

Req.22 The system should be able to identify deadlocked processes, kill them, perform cleanup and restart them. Whenever possible the event which caused the deadlock should be saved (see Fig. 9).

4.4 Monitoring and control

4.4.1 Starting the system

Starting the system should be possible by defining three parameters: the input source, defining PC to add to the system and which part of the system should be run on which machine.

Req.23 Starting the system should be preceded by specifying the input source, assigning PC's to the system and assigning tasks to these PC's (Fig. 10).

4.4.2 Stopping the system

In some situations it may be necessary to stop the system before end of file is reached. Depending on how critical it is to stop the system, two methods should be possible - hard stop and soft stop. While hard stop ends immediately loosing all the data that has entered the system but was not put out, the soft stop waits for the processing of the events that are in the system to be finished but stops reading new events immediately.

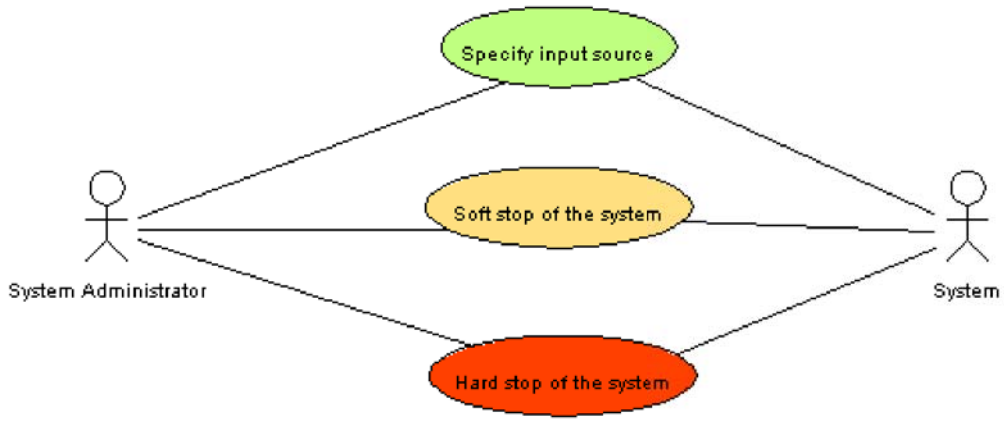


Fig. 10 . Start, stop use case diagram

Req.24 A hard stop method should be implemented which allows for stopping the system immediately. Loosing data is acceptable (Fig. 10).

Req.25 A soft stop method should be implemented which allows for stopping the system after processing of events that already are in the system is finished. Data loss is not acceptable (Fig. 10).

4.4.3 Monitoring interface

A basic monitoring interface is required which allows to see how much data has been already processed, what run is currently being processed, if the system has enough processing power to deal with the input data stream. More in depth monitoring is done at event level by the Fortran analysis program which creates histograms of the processed events so this functionality is not required by the system.

Req.26 An interface should exist allowing to check the amount of data that has been processed and the current position in the data flow (the run number) (see Fig. 11).

Req.27 It should be possible to check if the system has enough processing power to cope with the input and output data streams (see Fig. 11).

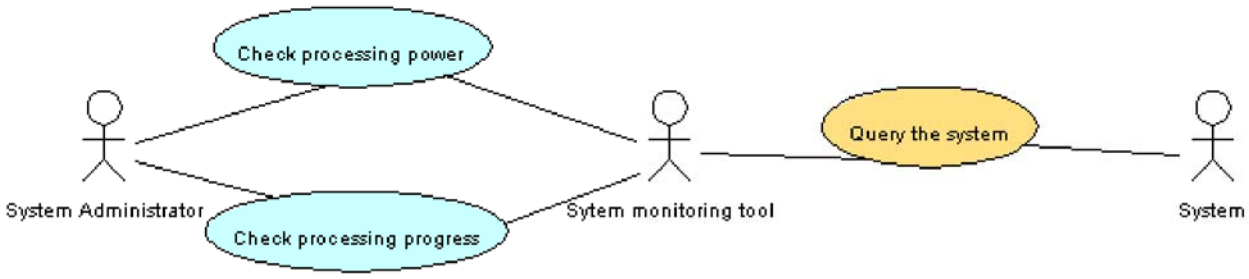


Fig. 11 . System monitoring use case diagram

4.5 Extendability

4.5.1 New event types

Currently the support for FPACK Events is necessary but in future other data format may be used in H1. The system should be as format independent as possible and allow for an easy switch to object oriented data models.

Req.28 System design should allow for an easy switch to new event formats.

4.5.2 Additional modules

The system should be built in a modular way allowing new modules to be easily added - like system controlling and monitoring tools. These additional modules may be possibly written in other programming languages than that used for the core system components.

Req.29 Modular design of the system allowing new modules to be added easily. New modules may be written in other programming languages than the core system components.

5 Design

5.1 Selecting the communication model

Selecting the model for interprocess communication in the system was an important part of the design since existing models vary greatly and can have great influence on the system. After many discussions (a comparison of different solution can be found in [17]) the CORBA standard has been chosen. It allows for:

- An object oriented model.
- Communication defined by specifying object interfaces and simple data structures (unfortunately no object-by-value support in C++ yet)

Other advantages of CORBA (a bit more related to implementation than design) are:

- A well defined, wildly supported standard with many open source implementations,
- Operating system, machine and location transparency,
- Bindings to all modern programming languages: C, C++, Java, Python among others,
- Request handling policy which creates a separate thread for each request.

See the CORBA standard home page for more details [18]. This choice was influenced by Req. 29.

5.2 General system design

The basic unit in the system is a node which is equivalent to one PC. We want to design the system in such a way that a node contains a single operational unit. These units can be divided into four groups – input nodes, output nodes, worker nodes and controlling nodes (In Fig. 12. only two input and output nodes, and six worker nodes are shown but the system is scalable). Input nodes are responsible for reading input data and make them available to worker nodes. Worker nodes acquire data from input nodes through polling, perform some kind of computation on it and pass the results to output nodes (pushing). Output nodes move the data they get from worker nodes to some place outside the system. The data flow is: input-> worker nodes -> output. Controlling nodes lie outside the data flow and have asynchronous access to the system nodes. There are two controlling nodes – the barrier server and the Controller. The barrier server is responsible for input synchronization and the output-input callback mechanism, while the Controller takes care of the whole system by starting, stopping and monitoring the nodes.

Input, output, and worker nodes have special data buffers (described in chapter 5.3.2). Input

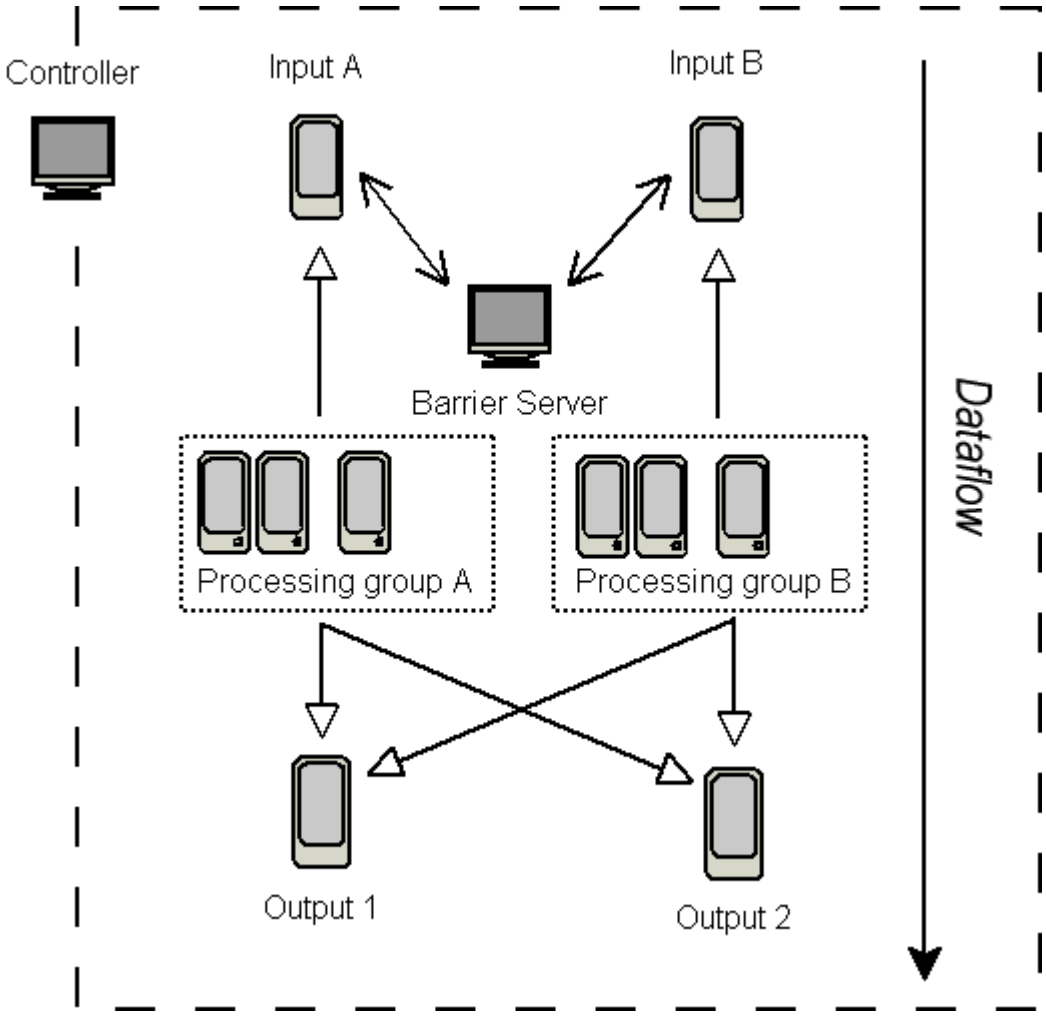


Fig. 12. General scheme of the system.

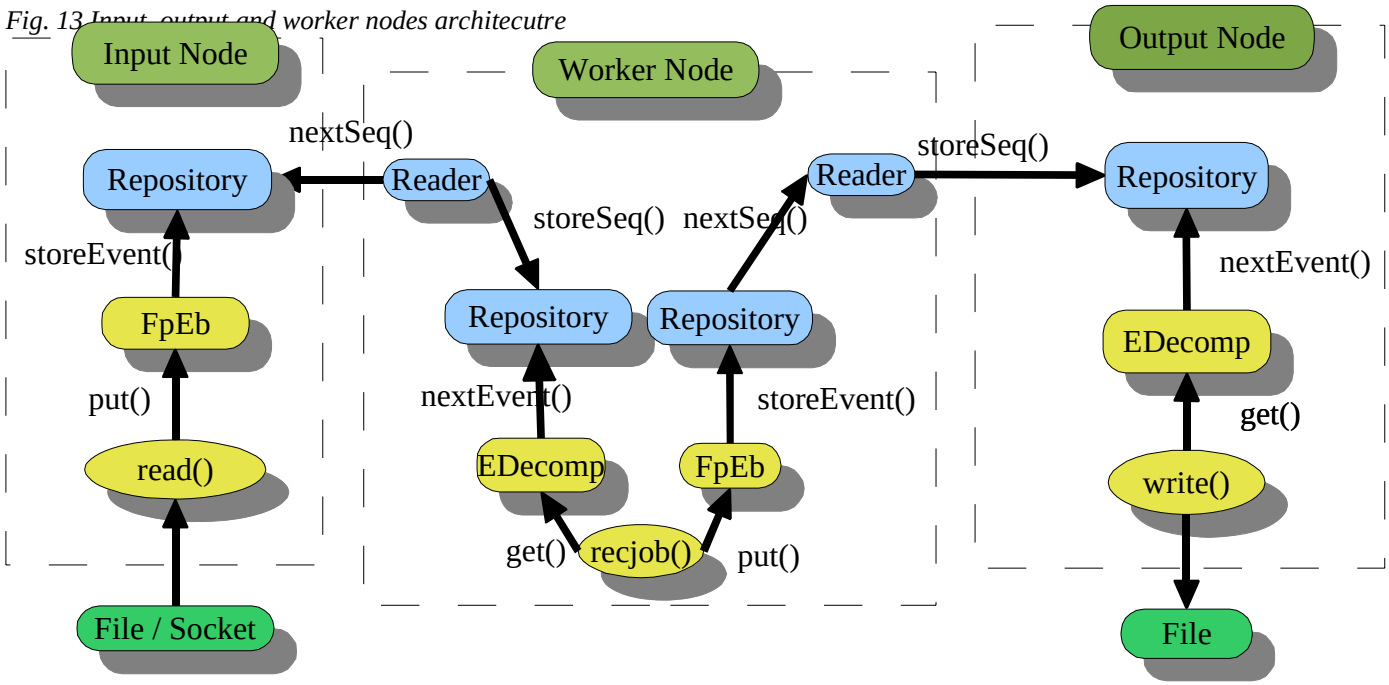
nodes buffer data they read and make them available to worker nodes. Worker nodes have two buffers, one for data read from input nodes and one for the results which are buffered before being sent to the output. Output nodes have one buffer for data they get from worker nodes.

In Fig. 12 a limitation of the system is visible, which was made to simplify the design - each worker node can write data to multiple output nodes but it can read data only from one input node. Thus worker nodes can be grouped based on the input node they read from (Processing group A and B in Fig. 12).

5.2.1 Architecture overview

A more detailed view of the system, focusing of input, output and worker nodes structure, can be seen in Fig. 13.

Fig. 13 Input, output and worker nodes architecutre



The input nodes consists of three main elements. A routine for reading data from a socket or file exists. The class *fpEB* (see 5.4.3) creates CORBA structures from that data. These structures are stored in a buffer called event repository (see 5.3.2). It implements a CORBA interface to make events available for remote readers.

Worker nodes are more complicated. At the input side of a worker node an instance of the class *Reader* is responsible for acquiring data from the input node. Data from the input node is buffered in a local input repository. The *EventDecomposer* (see 5.4.2) reads events from the repository and feeds data into the reconstruction or analysis program of the node. Results are gathered by an *fpEB* and put into the local output repository. Another *Reader* takes the results from the local output repository and stores them in the output node.

The output node is constructed analogically to the input node. A repository to buffer results coming from worker nodes exist. Events are read from it by an *EventDecomposer* and fed into a function which writes them to file or other storage.

In the following chapters a bottom-up view of the system design will be presented. Starting from event representation, through main system modules, to objects which represent different node types.

5.3 Events and the data flow

5.3.1 Events

An event is a set of computationally independent numerical data. At input side data arrives in FPACK format. In order to transfer events between machines or processes we have to wrap them into CORBA structures. Special events called barriers are introduced to allow for distribution of calibration constants, grouping of events and marking the end of data flow.

Physical, FPACK and CORBA events. The data arrives at the input side as physical records. A facility is needed which would extract parts of logical records from them and create FPACK Events (Req 7).

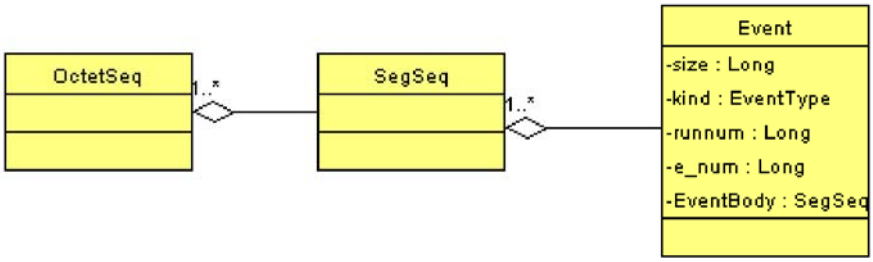


Fig. 14 . Event class diagram

After separating out FPACK events we are ready to distribute them among worker nodes. Because we have decided to use CORBA for interprocess communication, we have to wrap them into CORBA structures first. This CORBA structures will be referred to as “CORBA Events”. CORBA Events contain all the data of the FPACK Event kept as a sequence of logical record parts - *EventBody*. Each logical record part, represented by the *SegSeq* structure (see Fig. 14), is in turn a sequence of octets. The structure contains also some fields with basic information about the event, like event type, event size, run number etc. This information is in the FPACK data but it can not be accessed without unpacking of the FPACK Event, hence for performance reasons it is duplicated. However this information is used only by modules which deal with the FPACK structure of the event (*fpEB* and *EventDecomposer*). At other modules only the size parameter of CORBA Events is being used. This restriction has been made to allow for an easy switch to new event formats as mentioned in Req. 28.

Event life time. Events in the system exist either as FPACK Events or CORBA Events. On the input side we get events in FPACK format, but before we can send them to other processes each event has to be wrapped into a CORBA Event. Unfortunately CORBA Events are not understood

by the analysis and reconstruction programs. FPACK Events have to be extracted from CORBA Events before any processing can be done. The result of the processing is also in FPACK format so

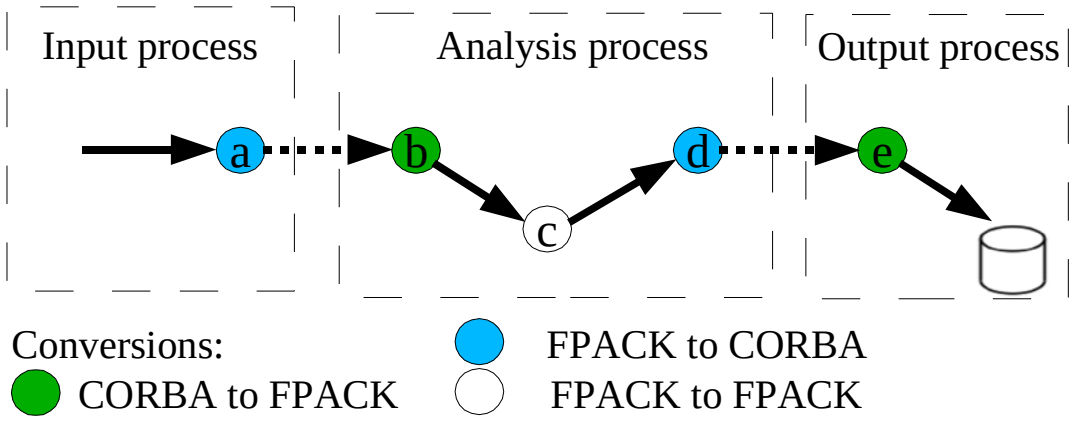


Fig. 15. Event life time

we have to create CORBA Events from it again before sending them to the output. On the output side FPACK Events have to be extracted once more before they are written to tape or disk. The lifetime scheme is shown in Fig. 15.

The object responsible for CORBA Event creation is called *fpEB* (FPACK event builder) and its design is described in chapter 5.4.3. Unpacking of FPACK events from CORBA events is done by *EventDecomposer* which in turn is described in chapter 5.4.2.

Barrier events. An ordinary event is removed from the input side when sent to an analysis process. However Reqs. 15-18 require that some data is sent to all worker nodes, not only one. To achieve this special events called barriers were introduced. A barrier is not removed from the input side until all worker nodes have read it, and also can not be removed from the output side (written to disk) until all worker nodes have output it. They can be used to distribute calibration constants, to separate groups of events or to mark the end of data flow. How this is achieved will be described in more detail in the following chapters. Barriers were first proposed by Alan Campbell [19].

Barrier representation. Barriers, like ordinary events, exist on FPACK and CORBA level. On CORBA level the same structure '*Event*' is used. However the fields in the structure have different meaning for a barrier than for an event. Events and barriers can be distinguished by a special value in the CORBA structure 'kind' field and a special value in FPACK record header.

5.3.2 Event repository

In all places in the system where events have to be buffered, they are kept in a special data

structure called event repository. The event repository is one of the most important parts of the system as it regulates the transfer of events between different processes. It implements a CORBA interface to make itself available to remote calls. Events in the repository are kept in sequences to optimize network transfer when remote store and get calls are made. However local store and get calls may operate on single events. As multiple readers and writers may access the repository at the same time, all functions need to be threadsafe. The repository is also responsible for appropriate handling of barriers. This will be described in chapter 5.3.3.

Further, repository readers and writers will be referred to as repository 'clients'.

Repository configuration. The repository can be configured by passing arguments to the constructor. The main parameters are:

- Maximum size (in bytes) of a sequence, that can be stored in the repository (if an event exceeds this limit it will be stored anyway in one-element sequence).
- Maximum size of the repository in sequences. This value multiplied by the previous one gives an approximation of the total repository size. This is an important parameter since we don't want to go past the memory available on the PC and start using swap. This would hit performance badly or even cause the system to crash. However one must be careful - the maximum sequence size limit may be exceeded as described above, and also the maximum number of sequences can be exceeded by one (see chapter 5.3.3 “The 'stuck' situation” for details).
- Maximum sequence length. When events are small this limit may be useful so that not too many events are stored in a single sequence, as sequence is the basic data transfer unit between nodes.
- The number of input nodes - this is an important information for multiple input support to work. This will be discussed it in chapter 5.3.3 “Multiple input support”.

Storing and reading events from the repository. Single events are usually small about 100kB, and making a CORBA call for each event would slow down the transfer. Therefore events are kept in repositories grouped in sequences. Repository clients making remote calls obtain and store sequences of events by making *nextSeq*() and *storeSeq*() calls respectively (Fig. 16). The sequence size may by configured to efficiently utilize the network. The *Reader* class is designed to transfer sequences between two repositories (see chapter 5.4.1). All functions which can be seen on the class diagram implement the *Repository* CORBA interface.

Local clients, like FPACK event builder or event decomposer, which are in the same process or at least on the same machine, can read and write data to the repository one event at a time. *storeEvent*() and *nextEvent*() calls are used for this.

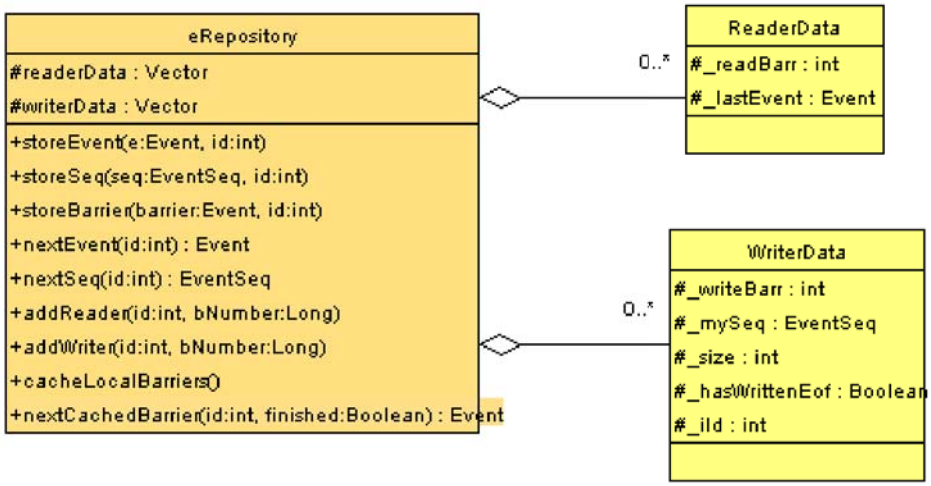


Fig. 16 . Repository class diagram with client data classes

Reading and writing algorithm will be described in more detail in chapter 5.3.3 “Algorithms for accessing the repository”.

5.3.2.1 Reader and writer data

Each call to the repository includes an 'id' argument. It is a number identifying the client of the repository. The repository keeps a list of records with information about each client (Fig. 16).

The *_writeBarr*, *_readBarr*, *_hasWrittenEof* members of the *WriterData* and *ReaderData* structures, equal to the number of the barrier last written by a writer, the number of the barrier last read by a reader and if the writer has written the EOF barrier respectively. This information is used to clean up after the client when he is removed. It has to be kept inside the repository in case the client process crashes and the client can not supply this information himself.

The variables *_size*, *_mySeq* and *_lastEvent* are used only when the client uses *nextEvent*() or *storeEvent*() functions. In *_lastEvent* the event returned by the last call to *nextEvent*() is stored (see 5.3.3 “Algorithms for accessing the repository - *nextEvent*()”). Each writer storing events with *storeEvent*() is building his own sequence which is kept in *_mySeq*. When the *_size* reaches the limit for sequence size, the sequence is stored with *storeSeq*() (see 5.3.3 “Algorithms for accessing the repository - *storeEvent*()”).

Repository structure. The repository is composed of two layers. On the higher layer, which is the *eRepository* class, synchronization of the calls from clients is performed. The repository operates with mutexes and conditionals to ensure threadsafety. The lower layer is a data structure which can be accessed only sequentially. It directly implements the barrier algorithm as described

in chapter 5.3.3 “Storing and retrieving barriers”. The structure is called *BarrierList* (see Fig. 17) and is basically a FIFO implemented using a list. Only barriers are treated in a different way. The list elements derive from class *ListElem* and can be either *OrdElem* which encapsulates a sequence of events, or *Barrier* which encapsulates an event which is a barrier.

The *storeSeq*(), *storeBarrier*(), *nextSeq*(), *nextEvent*(), *addReader*(), *addWriter*(), *writerNotification*() of the repository make calls to corresponding functions in the *BarrierList*. On

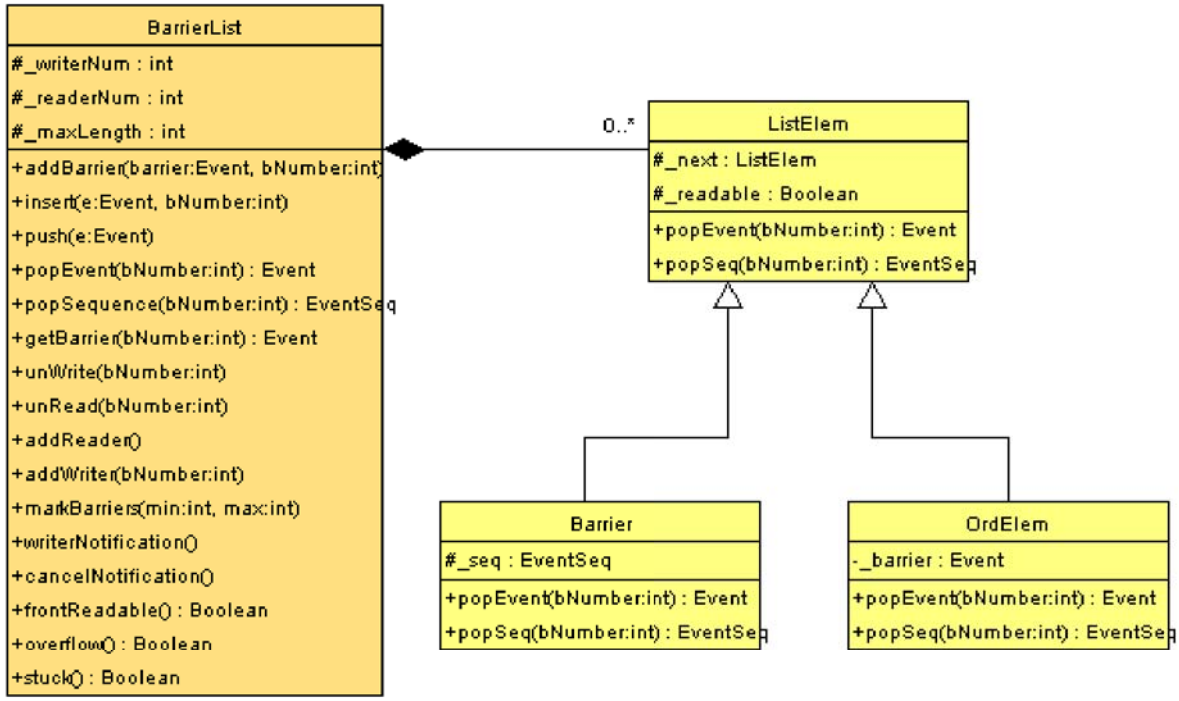


Fig. 17 . Barrier list class diagram.

the repository level synchronization locks are acquired and released, checks are performed if the operation can be completed or if the thread has to wait on a conditional. On the barrier list level modifications to the list are done, no checks are performed and it is assumed that the operation can be completed.

In chapter 5.3.3 the algorithms behind these functions will be briefly described.

The *longPing*() function. The function returns a snapshot of the repository state. It returns a structure containing information about number of events and sequences that were stored in the repository and removed, the current highest and lowest barrier number, number of sequences in the repository and some additional information (see Fig. 18). This function is an implementation of the CORBA interface and can be accessed by a monitoring program to display information about the repository (Req. 26) and to see if enough processing power has been assigned for the current tasks (full repositories will indicate that it hasn't) (Req. 27). When the *longPing*() function returns the same value for some time it may also be an indication for a deadlock which is part of Req. 22.

5.3.3 Barrier handling

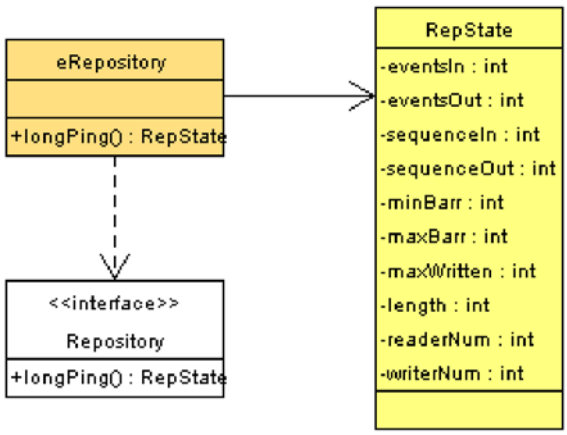


Fig. 18 . Repository state class diagram.

Barrier tasks. There are three main barrier tasks. The first one is to separate groups of events. We need this to separate events from different runs. Barriers called mark barriers can also be inserted without reason just to mark how far is the processing. When a barrier is written at the output we know that all events up to that barrier have been processed and either rejected or successfully reconstructed.

The second task is, as it was already mentioned, to distribute calibration constants. Data that comes with constants barriers has to be distributed to all nodes and we must be sure that it reaches each worker node before the events that are after it in the input data stream.

The last task to which barriers can be applied is to steer the framework. There is currently one such barrier – the End Of File barrier. Each nodes ends after reading this barrier.

Storing and retrieving barriers. There are two main requirements the barriers have to fulfill:

1. Each barrier has to reach all worker nodes (Reqs. 16-19).
2. Events from between two barriers have to stay there all over the data flow Req. 15.

This can achieved by making following assumptions about barriers:

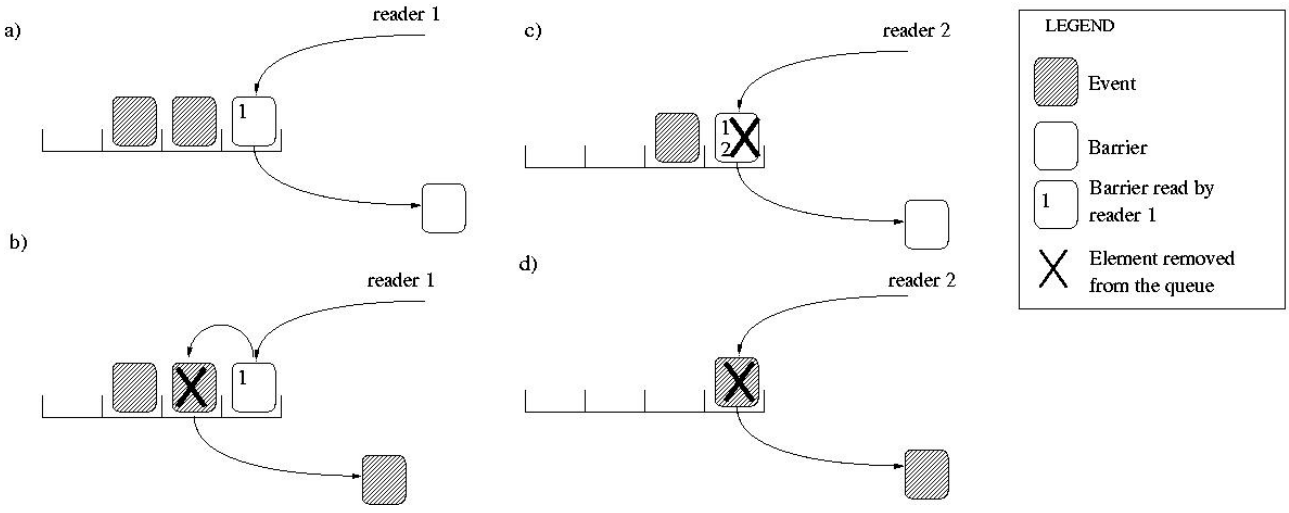


Fig. 19. Barrier list with two readers.

1. Events that are behind a barrier in a repository, can not be read until the barrier has been made

readable (see Fig. 20).

2. A barrier is made readable when all writers have written it (see Fig. 20 and Fig. 21).
3. When storing events a writer has to know which barrier he has written last, and store events after this barrier and before the next one (see Fig. 21 c,d).
4. A barrier is not removed from the list until all readers have read it (see Fig. 19 a,c and Fig 20 d).

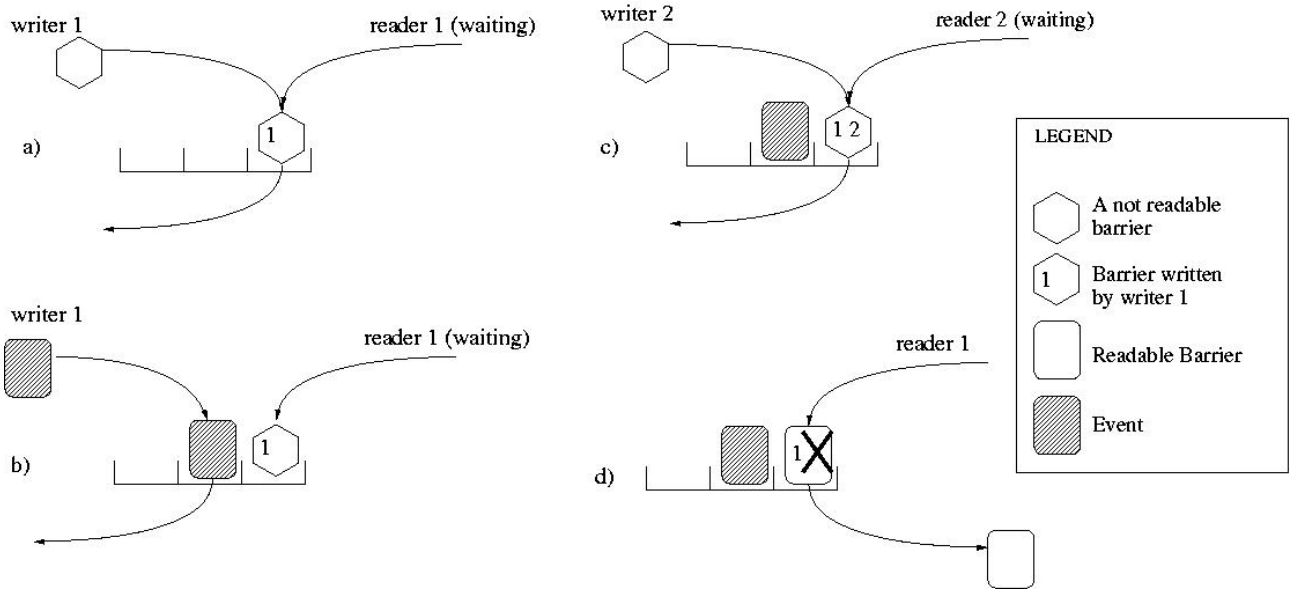


Fig. 20. Barrier list with one writer and two readers.

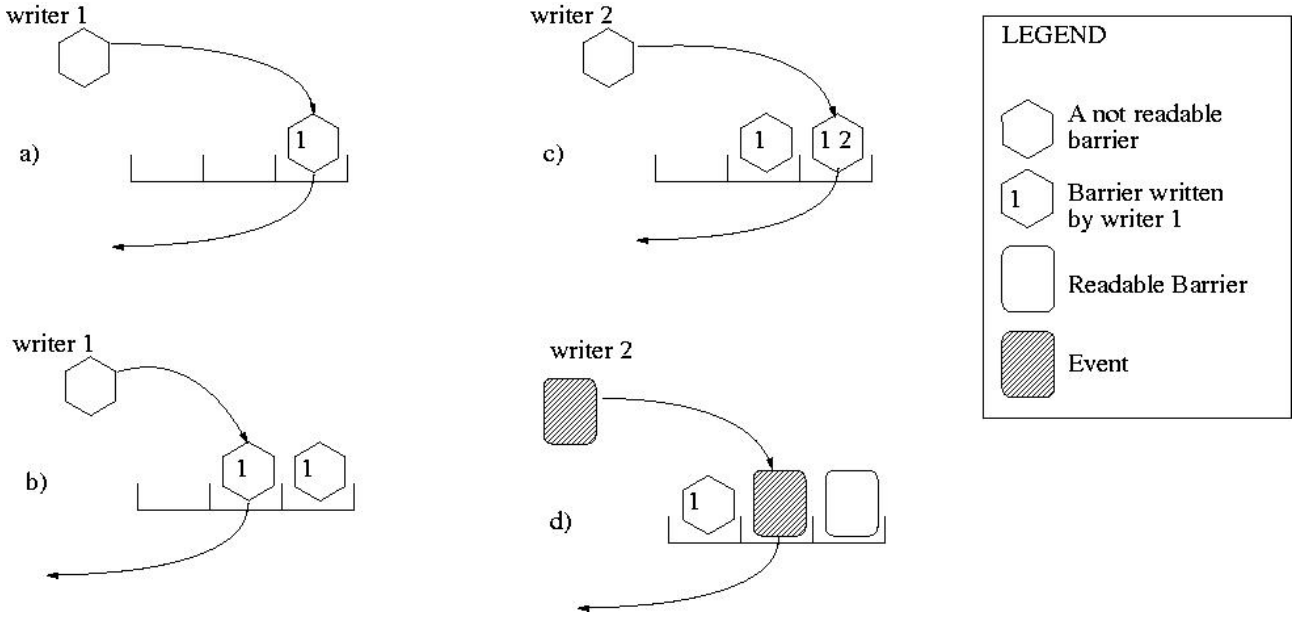


Fig. 21. Barrier list with two writers.

We can see that assumption 4. implies request 1. Assumptions 1-3 imply request 2. To improve performance we can make an additional assumption:

1. A reader which has read a barrier can read events that are behind this barrier even before the barrier is removed from the list (see Fig. 19 a, b).

To make barrier management possible each barrier in the system is given a unique number. The barrier number is increased by one for each new barrier.

Checking the barrier list status. There are three functions which allow readers and writers to see if they can write to or read from the list. The *frontReadable*() function returns true if the first element of the list is readable (it will return false if the first element is a barrier which has not been written by all writers). *overflow*() will return true if the list is true. That is when the maximum number of sequences the repository is configured to store has been reached.

The 'stuck' situation. The function *stuck*() will return true when !*frontReadable*() *&& overflow*(). When can such situations occur? Let's assume we have writer 'A' and writer 'B'. 'A' is fast but 'B' is slow. Now 'A' has written a barrier to the repository and after that so many sequences that the repository limit was reached. Nothing can be read from the repository because the barrier that is at the front is not readable since 'B' has not written it yet. In this situation *stuck*() will return true. Now if 'B' writes the barrier everything is fine. But if 'B' is so slow that it still needs to write some old sequences **before** the barrier, we have a deadlock - 'B' wants to write a sequence but the repository is full and nothing can be read from it. In such situation 'B' will be allowed to insert his sequence, breaking the limit condition but removing the deadlock.

You might notice that the limit will never be exceeded by more than one sequence, since after inserting a sequence at the front of the list, *frontReadable*() will return 'true' and the 'stuck' situation is not valid anymore.

Algorithms for accessing the repository. There are four functions which allow to access data in the repository: *nextEvent*(), *nextSequence*(), *storeEvent*(), *storeSequence*(), *storeBarrier*(). The functions can be accessed by many readers and writer simultaneously and have to be threadsafe. Two conditional objects and one mutex ([20],[21]) are used for this. Writers wait on the *overflow_condition* when the repository is full and can not accept any new sequences. When a reader removes a sequence from the repository making room for writers, it calls broadcast on this conditional to wake up all writer threads waiting. Analogically readers wait on *empty_condition* when there is nothing to be read in the repository and it is broadcast by writers when they store a sequence or make a barrier readable. The algorithms behind these functions will be briefly described.

The function *nextSequence*()

```
1 lock mutex,
2 get the ReaderData structure
```

3 **while** !*frontReadable*()
  3.1 **if(** *stuck*() **)** *broadcast*(*overflow_condition*)
  3.2 **wait**(*empty_condition*)
4 call *popSequence*(*bNumber*) on the barrier list (*bNumber* is taken from *ReaderData*)
5 update *bNumber* if necessary (if a barrier has been removed)
6 **if** a barrier has been removed call *cacheLocalBarriers*()
7 *broadcast*(*overflow_condition*)
8 **unlock** mutex
9 **return** sequence

In point 3.1 we have a 'stuck' situation as explained in chapter 5.3.3 “The 'stuck' situation”. Broadcast is called because the writer who can make the front barrier readable may be waiting on a conditional. In point 4 and 5 the *bNumber* is used. It is the number of the last barrier this reader has read. This information is passed to *popSequence*() so that the same reader does not read the same barrier twice.

The Function *nextEvent*(). *nextEvent*() is very similar to *nextSeq*(). The differences are:

- instead of *popSequence*(), *popEvent*() is called - this removes an event from a sequence at the front of the list, or a barrier is read.
- broadcast(*overflow_condition*) is called only if a sequence becomes empty and can be removed from the list.
- store a copy of the removed event in the *ReaderData* structure (this replaces the previous event copy).

The function storeSeq().

1 **lock** *mutex*
2 get the *WriterData* structure
3 **while** *overflow*()
  3.1 **if**(*stuck*() *&& bNumber == minBarr*) **break**
  3.2 **wait**(*overflow_condititon*)
4 call *insert*(*event, bNumber*) on the barrier list
5 *broadcast*(*empty_condition*)
6 **unlock** *mutex*

In case of a writer *bNumber* is the number of the last barrier written. In point 3.1 we have the 'stuck' situation again. *minBarr* is the number of the last barrier made readable. If *bNumber* == *minBarr* the conclusion is that the next barrier this writer is going to write is the first unreadable barrier - the barrier at the front of the list. So letting this writer write it's sequence will 'unstuck' the list.

The function *storeEvent*(). Storing events with *storeEvent*() results in a sequence being build out of them. This sequence has a size limit set in the repository object construction. However when a barrier is encountered it can not be put into the sequence. So when a barrier is found the sequence is stored even if it's not yet finished, the barrier is stored with *storeBarrier*() and a new empty sequence is created.

1 **lock** *mutex*
2 acquire *WriterData* (contains a sequence under construction)
3 **unlock** *mutex*
4 **if** adding the new event to the sequence would exceed the size limit (and the sequence has non 0 length)
    4.1 store sequence with *storeSeq*()
    4.2 create a new empty sequence in *WriterData*
5 **if** event is not a barrier, add event to sequence
6 else
    6.1 store sequence with *storeSeq*(), create new, empty sequence
    6.2 store barrier with *storeBarrier*()

The function *storeBarrier*(). *storeBarrier*() calls *addBarrier*() on the barrier list and broadcasts *empty_condition* if a barrier was made readable.

Multiple input support. It was mentioned before, that a repository has to know the number of input nodes in the system. This is necessary because each input node gets a copy of all barriers. Now lets assume that worker nodes associated with one input node are initialized quickly. They write barriers to the output repository, and the output repository being unaware of other input nodes in the system allows that these barriers are made readable and are removed from the system. Now finally worker nodes associated with the second input node are started, they want to write barriers to the output repository which have been already removed from there and the system crashes.

To avoid such situation, each output repository has to know from how many inputs it receives data and will not allow barriers to be removed until at least one worker node associated with each of the input nodes will register to it.

The barrier cache. Some barriers contain data that changes the way events are being processed. When a new worker node (Req. 2) is added to the system, many barriers may have

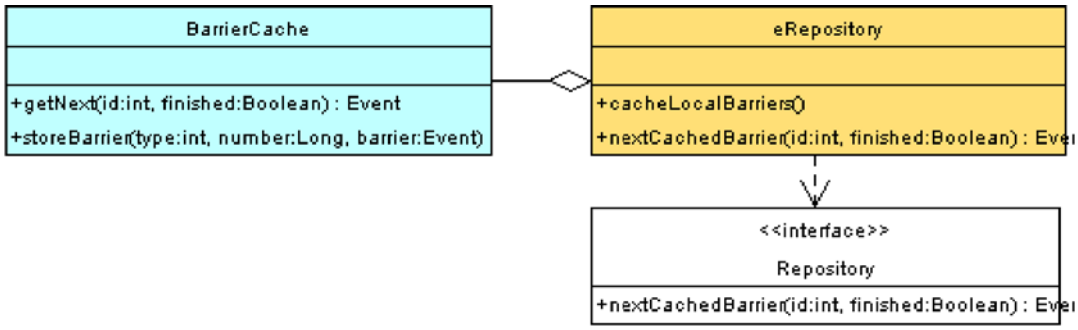


Fig. 22 . Barrier cache class diagram

already been removed from the input. Such node would have no access to these barriers anymore. To avoid such situations, the barrier cache has been introduced (see Fig. 22). It keeps all barriers relevant for events processing that have been removed from the input node. A new worker node has first to read and process all barriers from the barrier cache before starting to read from the input repository. This is done by subsequent calls to *nextCachedBarrier*(). The finished parameter of this function will be set to true when the last barrier is read. The process of caching and reading cached barrier is shown on sequence diagram in Fig. 23.

Cached barriers should not be introduced again into the data flow, they can not be output by the worker nodes but should be removed. They can be recognized by their *bNumber* which is lower then the lowest barrier number of the barriers that are still in the system.

Keeping the data flow consistent. The availability of barriers for new worker nodes is not the only issue we have to consider when adding new nodes. They should be added as smooth as possible without interrupting the overall data flow. That counts also for removing nodes.

Adding readers and writers. Each worker node is a reader to the input repository and a writer to the output repository. Adding a new worker node has to be done in three steps (see Fig. 24):

1. Inform the output repository that a new writer is coming. The writer number is increased and all barriers that are not yet readable will have to wait for this new writer to write them.

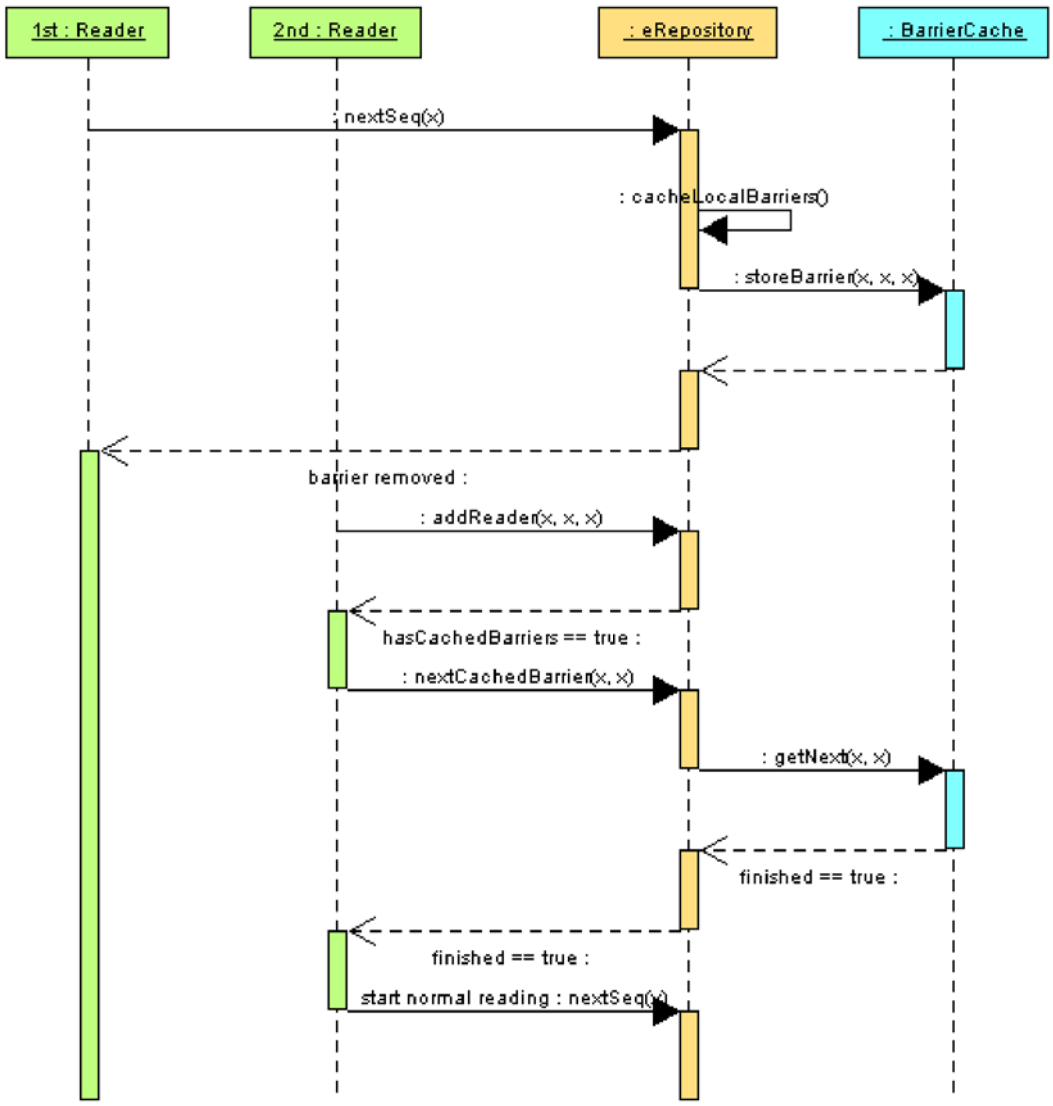


Fig. 23 . Reading cached barrier sequence diagram

2. Register in the input repository as a reader. The reader number is increased. No barrier will be removed from the input repository until it is read by the new reader. The number of the first barrier the new reader is going to read is returned.
3. Register in the output repository as a writer. Knowing the number of the first barrier we are going to read from the input repository we can register in the output repository. Barriers which are in the output repository and which we are **not** going to write, can be safely removed from the output repository (after all other writers have written them). The new writer has to wait for the last barrier he is not going to write to arrive in the output repository, before he can start writing.

We can also add a fourth step. This step can be executed in parallel with step 3. in a separate thread of execution:

4. Start reading from the input repository.
   a) Read all cached barrier from the input node

b) Start reading from the input repository

Removing readers and writers. Removing readers and writers is much simpler than adding. At the input repository all 'read' marks which have been left on barriers by the reader who is going to be removed have to be erased. At the output repository the same has to be done with 'written' marks.

5.4 Main system modules

5.4.1 The *Reader* class

The *Reader* class is actually a reader and a writer. It reads sequences from one repository and stores them in another. Both repositories are accessed through their CORBA interface. The class diagram is shown in Fig. 25. The reading/writing loop is done in a separate thread:

1. Register in the input repository - *registerInInput*() - if it has not been done earlier
2. Get cached barriers from the input repository and cache them in the output repository - *transferCachedBarriers*().
3. Register in the output repository - *registerInOutput*().
4. Read event sequences from the input repository and store them in the output repository until end of file is reached (*transferData*() function) or *markForStop*() is called. If barriers are attached to a sequence they are extracted and stored separately - *storeBarrier*(). Sequences are stored with *storeSequence*() function.
5. Store end of file in the output repository (if *dontSendEOF*() has not been called).
6. Unregister from the input and output repository - *unRegister*().

In point 1 of the algorithm the reader registers in the input repository. This is done with the *registerInInput*() function. The function is public and can be called from outside. To avoid double registration a boolean variable is set when the registration was done. Refer to the chapter 5.5.4 to see how this can be used. The *waitForStop*() function may be called by a thread to join the *Reader* thread and wait for it to finish.

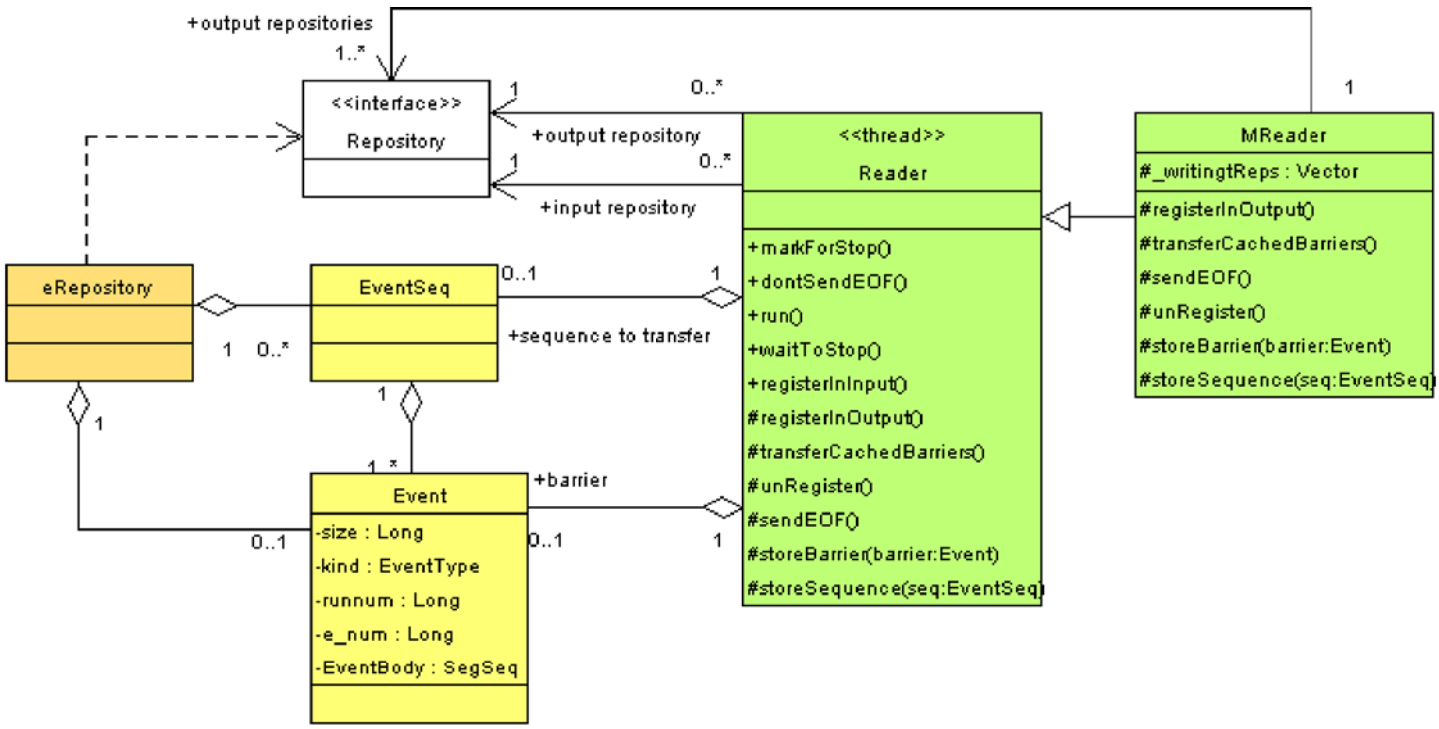


Fig. 25 . Reader class diagram

Multiple output support by *Reader*. The *MReader* class extends *Reader* by adding multiple output repositories support. Instead of a single output repository reference, a vector of references is kept by the class. All functions dealing with the output repository in *Reader* have to be overridden in *MReader*. The *storeBarrier*() and *storeSequence*() functions implementation defines how barriers and sequences are distributed among this multiple output repositories.

5.4.2 Event decomposer

The processing/reconstruction program is written in Fortran and requires data in FPACK format. The program calls *frspec*() function to acquire data (Req. 13). Event decomposer takes data from a repository one event at a time, and each time get() is called it extracts one record from the CORBA event and returns it. In the secure worker node design (see chapter 5.5.4) we want to be sure that a new event doesn't leave the repository before the last one has been stored in the output repository. Therefore *EventDecomposer* locks on the *wait_for_get*() function in *SynchronizedPipe* and waits for *fpEB* to call it before taking the next event from the repository.

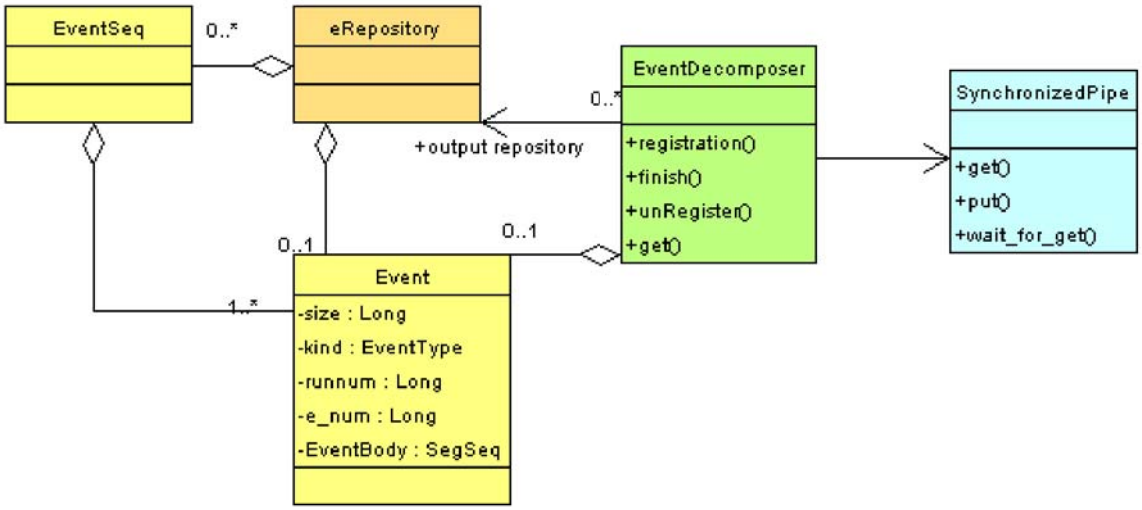


Fig. 26 . Event decomposer class diagram

5.4.3 Event builders

The FPACK event builder is factory of CORBA events. It reads FPACK records and produces CORBA events which are than stored in an event repository. The event builder exists in two kinds. *fpEB* works with worker nodes. It has simply to read the records and store events. The *fpEBOrbi* is associated with the input node. Besides transferring FPACK records into CORBA events it also has to insert barriers into the data flow and perform basic error checking.

The function responsible for creating CORBA events is run in a separate thread to allow for parallel reading of new records and/or sending ready events to other processes (Req. 3,4).

The *fpEB* class. The *fpEB* class (see Fig. 27) is associated with worker nodes. It gets FPACK records from the analysis/reconstruction program which contains the results of computation. It extracts FPACK Events from them, creates CORBA Events and stores them in the local output repository of the worker node. In this local repository events are buffered before they are sent to an output repository. The event building algorithm is following:

1. Register in the repository as writer.
2. Create a new, empty CORBA Event 'e'.
3. Get next physical record 'p' from the analysis/reconstruction program.
4. Extract next part of a logical record 'l' from 'p'.
5. **if** found an end of file barrier, send it to the repository and **goto** 8.

6. Store 'l' as a sequence of octets inside 'e'.
7. **if** the event is complete send 'e' to the repository with *storeEvent*(), then create a new, empty CORBA Event 'e'.
8. **if** reached the end of 'p' (of physical record) **goto** 2 **else goto** 3.
9. Unregister from the repository with *unRegister*().

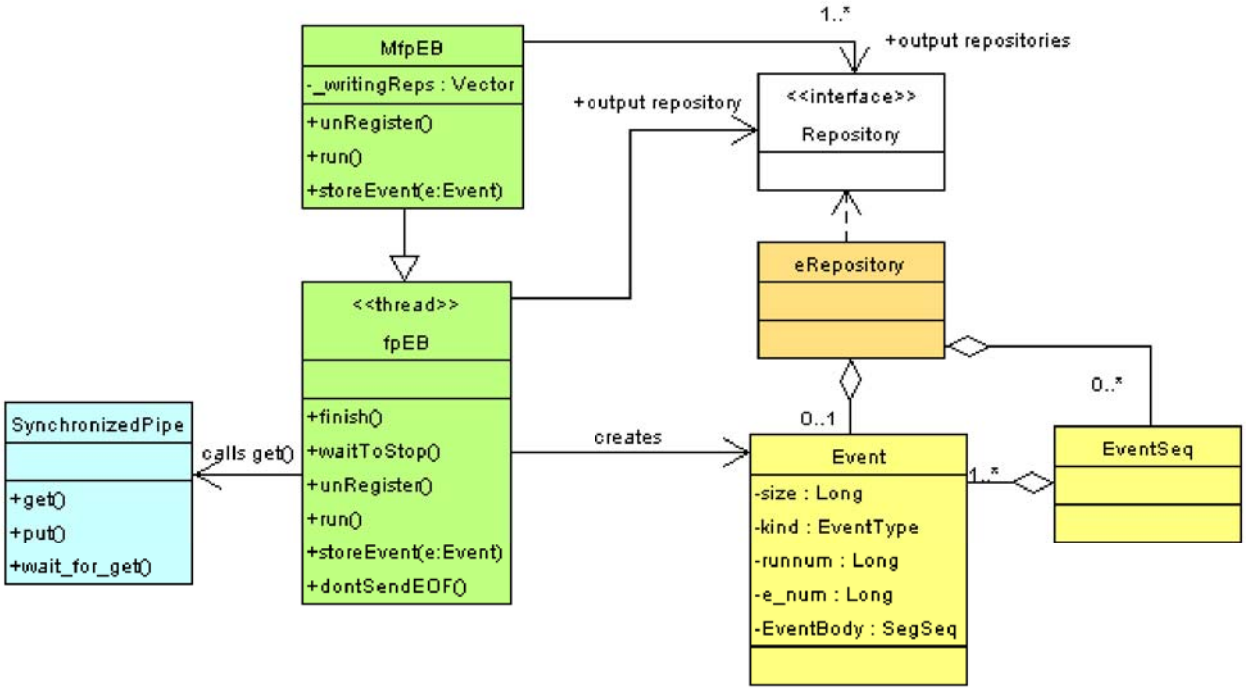


Fig. 27 . Event builder class diagram

The *fpEB* class implements a *waitToStop*() and a *finish*() function. The first one makes the thread, which calls it, wait until the *fpEB* thread is finished. The second function first tells *fpEB* to stop as soon as possible then it calls *waitToStop*(). As soon as possible means reaching a place in the building loop in which a variable is checked whether *fpEB* should stop or not.

The *MfpEB* class. The *MfpEB* class has been designed analogically to *MReader*. It extends *fpEB* by offering support for multiple output repositories. How events are distributed among multiple repositories is defined by the *storeEvent*() function implementation. *unRegister*() and *run*() functions have been overridden to handle a vector of repositories. *MfpEB* is used together with the *MMJobAdmin* class.

The *SynchronizedPipe* class. The class *fpEB* is getting data from the analysis/reconstruction program which (Req. 13), uses the *fwspec* function to output data. Because the program and *fpEB* building loop run in separate threads, a threadsafe buffer is required between them for passing data.

This buffer is called *SynchronizedPipe*. It offers a *put*() function for storing data and *get*() for retrieving it. It has a capacity of one data record.

The *fpEBOrbi* class. The *fpEBOrbi* (Fig. 28) class inherits *fpEB*. It is associated with input nodes and extends the functionality of *fpEB* by adding basic error checking of physical records, creating barriers and negotiating where to insert barriers into the data flow with barrier server. The communication with the barrier server is done via the *OrbiMan_i* class which implements the

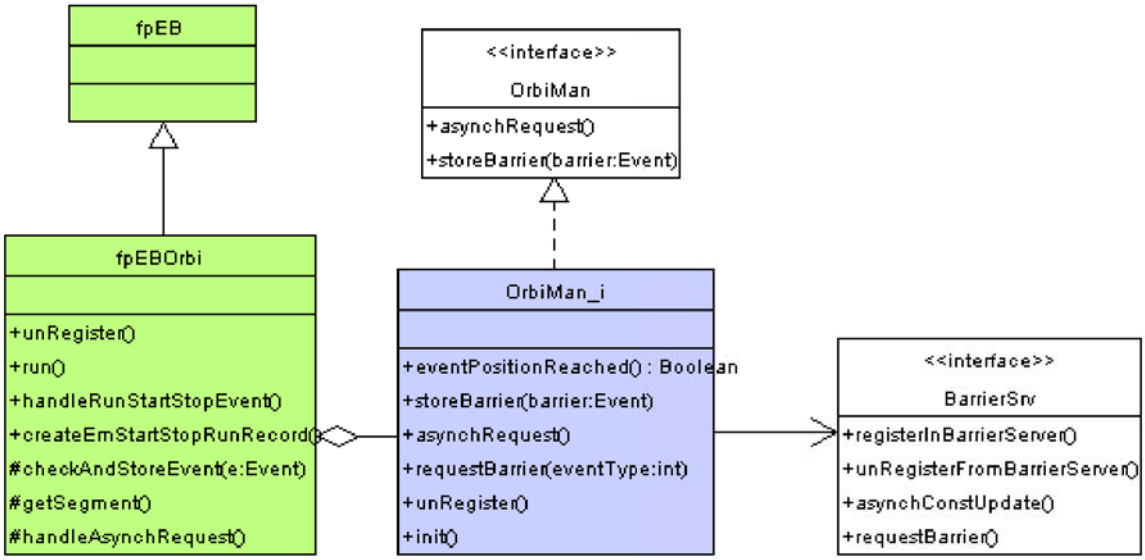


Fig. 28 . fpEBOrbi class diagram

OrbiMan CORBA interface.

The event building algorithm is following:

1. Register in the repository as writer.
2. Create a new, empty CORBA Event 'e'.
3. Get next physical record 'p' - *getSegemnt*().
4. Extract next part of a logical record 'l' from 'p'.
5. **if** found an end of file barrier, send it to the repository and **goto** 8.
6. **if** found a fatal error in record header delete event 'e', **goto** 2.
7. **if** found an asynchronous barrier request from barrier server call *handleAsynchRequest*().
8. **if** run number changed create run stop, run start, calibration constants barriers.
9. Store 'l' as a sequence of octets inside 'e'.
10. **if** the event is complete:
    11. **if** reached a specific event count number submit a marker barrier request.
    12. **if** reached a specific event number insert a barrier.
    13. send 'e' to the repository, create a new, empty CORBA Event 'e'.

14.**if** reached the end of 'p' **goto** 2 else **goto** 3.
15.Unregister from the repository.

Algorithm description. In point 6 of the algorithm a fatal error means that a logical record part header is corrupted making it impossible to determine where the logical record ends. This make the whole physical record useless and we have to throw it away. We might have also thrown away the beginning of the next logical record, so we will have to throw away its other parts which may come with the next physical record. This part of the algorithm fulfills Req. 20.

In point 8 two situations may lead to the creation of run start/stop barriers. In the first case a run stop and run start event will appear in the data flow. However it sometimes happens that run stop/start events of empty runs get lost. The algorithm has to check the run numbers of successive events. If for example after an event with run number 8, an event with run number 11 appears, run stop and run start events for runs 9,10 and 11 have to be created. The function *handleRunStartStopEvent*() is used to handle run start/stop events which appeared in the data flow, while *createEmStartStopRunRecord*() function is used when run changed but no run start/stop events appeared. The design details of these functions will not be described here, as they are based on FPACK format.

When run stop/start event is found or created a request to barrier server is made. This request contains that event. After run start a request for calibration constants is made by *fpEBOrbi*.

In point 11 a request for a marker barrier is made. An *fpEBOrbi* parameter determines how often such barriers should be inserted. For example setting this parameter to 100 means a marker barrier is inserted each 100 events.

There are four possible results of making a barrier request:

1. The request is accepted, a barrier is created and stored in the repository, *fpEBOrbi* may continue.
2. Another barrier request is accepted first and *fpEBOrbi* has to wait for its request to be accepted.
3. *fpEBOrbi* is told to continue creating events until a given event number is reached (in point 12), then the barrier is inserted.
4. The barrier is rejected, nothing happens, *fpEBOrbi* may continue creating events.

Handling of barrier requests will be described in more detail in chapter 5.4.4.

A barrier request may be also made from outside (not by an *fpEBOrbi*) or it may be specific to only one *fpEBOrbi*. For example each input node and hence each *fpEBOrbi* may be configured to insert a marker barrier after another number of events. In such situation the caller making the barrier request has to notify others about it. This is done by inserting into the data flow of all *fpEBOrbi*'s a fake event which is called an asynchronous barrier request. When *fpEBOrbi* finds such a request in the data flow (point 7), it makes a barrier request.

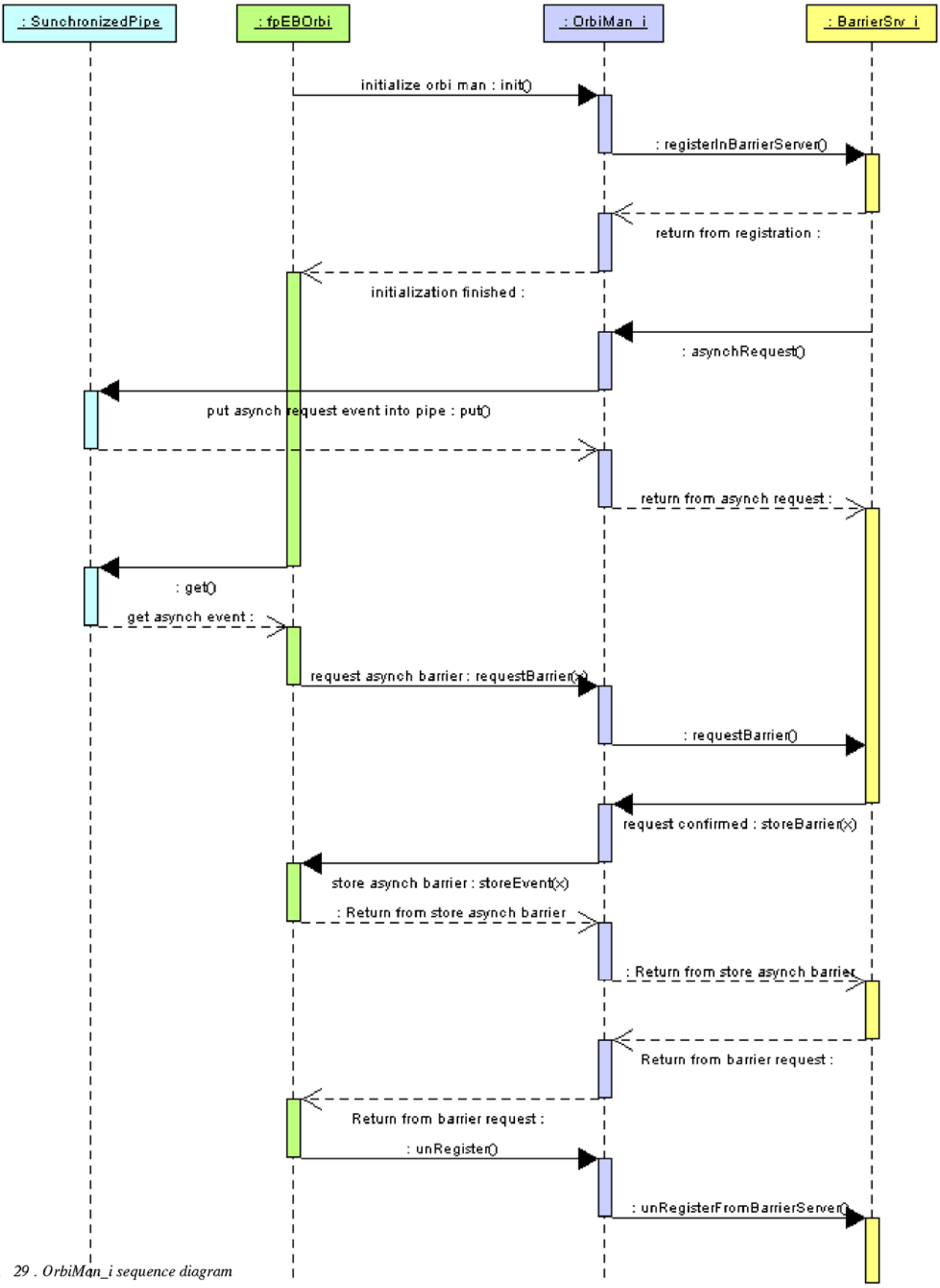


Fig. 29 . OrbiMan_i sequence diagram

Other functions. The *unRegister*() function has to be overridden in *fpEBOrbi* to include unregistering of *OrbiMan* from the barrier server.

A set of functions to perform basic error checking of events, writing diagnostic output when errors are found has been designed. Their description will be omitted here as it is tightly bound with the FPACK data format. The description of some other functions related to barrier building will be also omitted.

The *OrbiMan_i* class. *OrbiMan_i* is the class *fpEBOrbi* uses for communication with barrier server. It implements a CORBA interface - *OrbiMan* - allowing for bidirectional communication with the server. Sequence diagram in Fig. 29 shows the initialization of *OrbiMan_i*, the handling of an asynchronous barrier request, and the unregistering process. Synchronous barrier requests are made the same way. The only difference is that no asynchronous event is inserted into *SynchronizedPipe*.

5.4.4 Barrier Server

The barrier server (see Fig. 30) is responsible for synchronizing barrier insertion into the data flow. When the input stream is divided and more than one entry point to the system exists special precautions have to be taken, if we want that barriers are inserted at the same points.

Let's assume we have n entry points. The event numbers are unique and growing in the data stream. When a barrier is inserted at entry point n between two succeeding events x and y (x<y), it has to be inserted at all other n-1 entry points between two succeeding events a_i and b_i ($a_i < b_i$) where a_i<y and b_i>x for all i = 1 ... n-1.

To fulfill this requirement each insertion of a barrier has to be proceeded by a negotiation at which point in the data stream it should be inserted. All input processes have to take part in the negotiation and all input processes have to catch up with the process with the highest event number.

A state machine has been designed to decide what action should be taken based on the request that is in the front of the queue and the barrier type of the new request. Request from all input nodes have to be collected before a decision is made. An asynchronous barrier request is send to the input nodes if a request has to be forced. The barrier server communicates with the input nodes through an *OrbiMan* which is a CORBA interface. The barrier server has a list of references to all *OrbiMan*'s which corresponds to a list of all input nodes. First the state machine and actions that can be the result of making a barrier request, will be described, then the barrier server algorithm.

The barrier server is also responsible for barrier creation. The part of the design which has to

do with barrier creation has been left out as it is tightly bound with the FPACK format of the barriers.

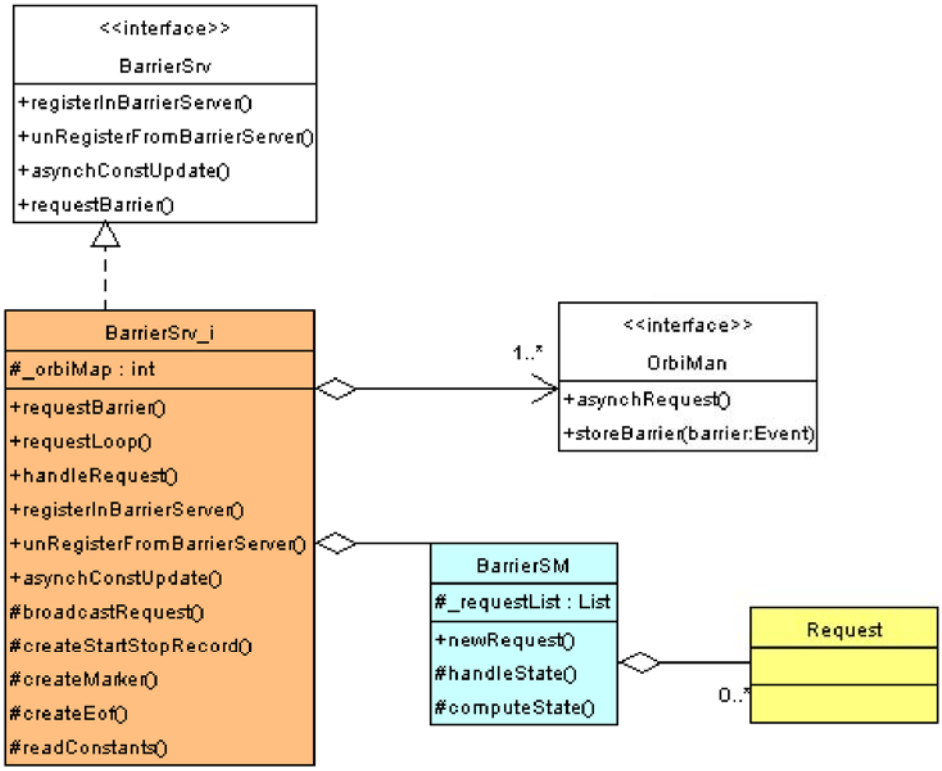


Fig. 30 . Barrier server class diagram

5.4.4.1 Barrier State Machine

The *BarrierSM* class is responsible for making a decision what action should be undertaken when a new barrier request is coming. This is done in two steps. First the state is computed with *computeState*(), then modifications to the request queue are made with *handleState*(). In Table 1 all possible states of the machine and corresponding actions have been put together. The first column contains the type of barrier that is being requested. The first row is the type of the barrier from the request queue which is being currently considered. The '+' and '-' signs mean that the barrier from the request queue has a greater (+) or a lower (-) run or event number than the requested barrier. When both barriers are Start/Stop or Em Start/Stop barriers the +/- signs mean run number, in all other cases they mean event number. The action symbols are described in Table 2, barrier types in Table 3. The last column in Table 1 contains barrier category which essentially describes how the given barrier appears in the data flow. Barrier categories are listed in Table 4.

Request	***Barrier from the request queue: type and run or event number.***																			
	EStart	+	-	Start	+	-	EStop	+	-	Stop	+	-	Cons	+	-	Mark	+	-	EOF	
Em Start	+	–>	<–	O	–>	<–	<–	–>	<–	<–	–>	<–	E	E	E	E	E	E	–>	A
Start	X	–>	<–	+	–>	<–	<–	–>	<–	<–	–>	<–	E	E	E	E	E	E	–>	A
Em Stop	–>	–>	<–	–>	–>	<–	+	–>	<–	O	–>	<–	–	<C	<W	–	<C	<W	–>	A
Stop	–>	–>	<–	–>	–>	<–	X	–>	<–	+	–>	<–	–	<C	<W	–	<C	<W	–>	A
Const	–>	–>	<–	–>	–>	<–	–	>C	>W	–	>C	>W	–	C	W	–	<C	<W	–>	F
Mark	–>	–>	<–	–>	–>	<–	–	>C	>W	–	>C	>W	–	>C	>W	–	C	W	–>	F
EOF	<–	–	–	<–	–	–	<–	–	–	<–	–	–	<–	–	–	<–	–	–	+	A
Nullbarrier	R	–	–	R	–	–	R	–	–	R	–	–	+	C	W	+	C	W	R	G

Table 1. State diagram of the barrier state machine

Sym	***Meaning***
+	The current request is confirmed.
->	Replace - place the new request at front of the queue, displacing the current one, confirmations are copied to the new request.
X	Remove - the current request is removed the new one takes his place, confirmation is copied.
<-	Append - the new request is checked against the next request in the queue.
O	Obsolete - the new request is ignored, the current one is confirmed.
W	Wait for all inputs to reach the same event position
C	Continue reading until the event position of the current request is reached.
R	Reject the request.
E	Error.
-	This situation does not occur (for example we do not compare run/event numbers when an EOF barrier request is made).
>C	Put the new request at the front of the queue and continue.
>W	Put the new request at the front of the queue and wait.
<W	Move to the next request in the queue compute the state once more and wait.
<C	Move to the next request in the queue compute the state once more and continue.

Table 2. Explanation of action symbols

Barrier Type	*Description*
Start	A barrier created at the start of each new run. It is based on a run start event which appears in the input data stream. It comes always after the run stop event (with exception of the first run).
Stop	A barrier created at the end of each run. It is based on a run stop event which appears in the input data stream.
Emergency (EM) Start	A barrier created at the start of a run when the run start event is missing. It comes always after the run stop event (with exception of the first run).
Emergency (EM) Stop	A barrier created at the end of a run when the run stop event is missing.
Calibration constants	A barrier containing the full set of calibration constants. It can appear after a run start barrier.
Update constants	A barrier containing changes to the calibration constants. It may come after a run start barrier or in the middle of the run. Update constants are treated in the same way as Calibration constants by the state machine, that is why there is no row with Update constants in Table 1.
Marker	A marker barrier. It may be inserted into the data flow at constant event intervals, for example each 100 events. It can be used to recover from crashes, to see how far the processing came.
EOF	A barrier created when an end of file appears in the data flow. It may be also created 'artificially' to make the system, or part of it, perform a clean close.
Nullbarrier	A barrier created as a result to an asynchronous barrier request. Such a request is made by placing a special event into the data flow. This event is recognized by the *fpEBOrbi*.

Table 3. Barrier types

A	Barriers generated automatically by all input nodes when a special event is encountered in the input data stream.
F	Barriers which don't appear in the input data stream - asynchronous barriers. Input nodes have to be forced to make requests for these barriers.
G	This is the Nullbarrier. It is created as a replay to an asynchronous barrier request. They are used only to confirm other barrier requests.

Table 4. Barrier categories

5.4.4.2 Barrier server algorithm

The barrier server is run as a separate process. Its main thread is running in a simple loop (*requestLoop*() function) awaiting for a request to be confirmed and then distributing the confirmed barrier (*handleRequest*() function) to all input nodes via *OrbiMan* references. Requests are made by input nodes by making a CORBA call to the *requestBarrier*() function. For each CORBA call a new thread is created so *requestLoop*() and *requestBarrier*() have to be threadsafe. The algorithm for handling incoming requests is following:

1 **lock** *mutex*.
2 **if** the request is new or it is different from the request at the front of the request queue:
  2.1 Call *newRequest*() on the state machine.
  2.2 **if** the request has been rejected **return**.
3 **if** the request is new:
  3.1 **if** the request should be forced - broadcast an asynchronous barrier request.
  3.2 **if** all *OrbiMan* have entered this function wake up all threads **else** suspend this thread.
  3.3 **if** the request has been forced, and the event number of the current request is lower than that of the request at the front queue, this *OrbiMan* has to leave the function and continue event reading until the appropriate event number is reached. When it will enter the function again its request will not be 'new' anymore (point 2 and 3)
4 The request is confirmed.
5 The thread is suspended until all other inputs have confirmed the request.
6 The request is confirmed - the main thread of the barrier server is woken up. All threads making a request are suspended until the main thread is finished with distributing barriers.
7 **unlock** *mutex*

5.5 Node types

5.5.1 The *Administrator* class hierarchy

There are three (not counting controlling nodes) categories of nodes in the system: input nodes, output nodes and worker nodes. There are several possible implementations in each category and it is possible that in future new additional implementations will be required. Each node is a multithreaded CORBA client or server. To make maintenance easier, speed up development and ease the addition of new categories or implementation in a category, a class hierarchy of so called administrators has been developed (see Fig. 32). An administrator is responsible for the correct

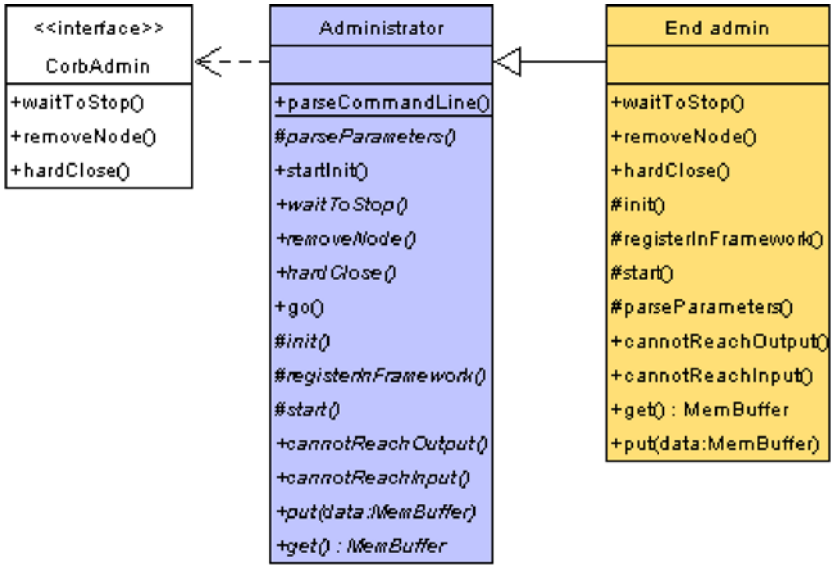


Fig. 31 . Administrator class diagram

order of initialization of all modules for a specific node type, for correct order of stopping and cleanup. The *Administrator* class which is at the top of the hierarchy offers a common interface to control different types of administrators both through the CORBA interface for remote access but

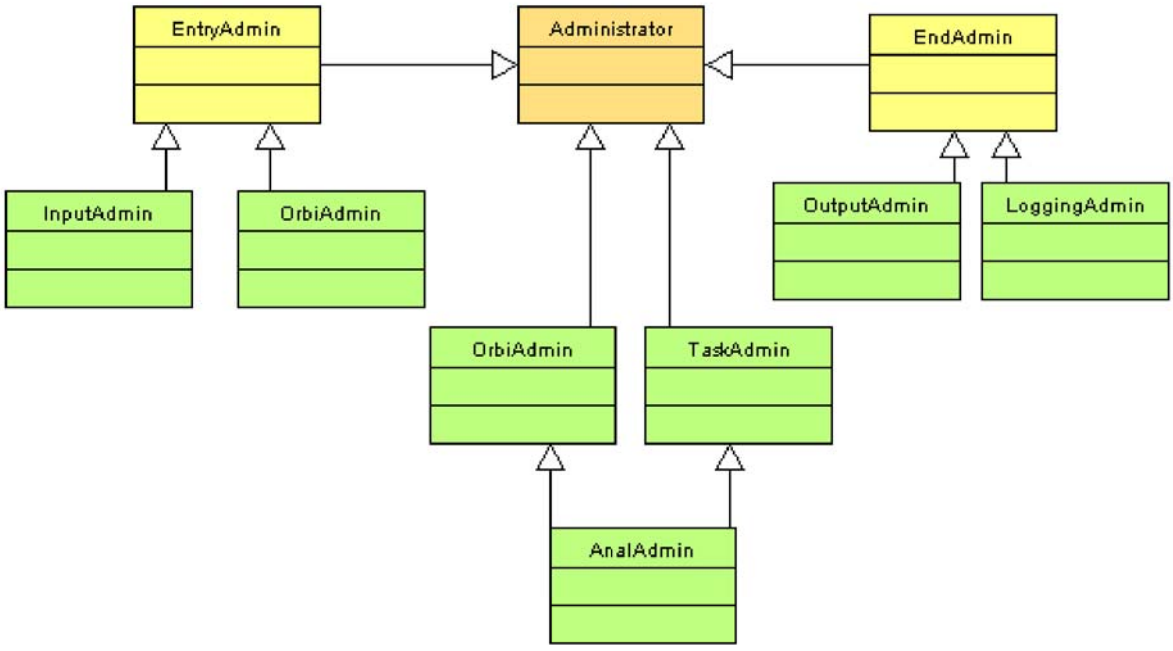


Fig. 32 . Administrator hierarchy class diagram

also through abstract classes which are then implemented by classes which derive from *Administrator*. This allows also local objects to access their *Administrator* in a way independent of the administrator type they are 'in'. You can see the abstract interface in Fig. 31. The *EndAdmin* class is an example of how the functions can be implemented.

5.5.1.1 Choosing the node type

We have decided that functionalities of different nodes should be kept in a single executable and that the exact node type should be selected at startup by choosing a concrete administrator for that node. A command line argument containing the name of the administrator type is passed to the program.

5.5.1.2 Starting an administrator

The startup algorithm which is common for all classes deriving from *Administrator* is following:

1 Read and parse generic command line arguments (*parseCommandLine*() function)
   1.1 **if** found the administrator type create an instance of that class **else** print an error message and **exit**.
2 call *startInit*()
   2.1 Read and parse specific command line arguments for the selected administrator class (*parseParameters*() function).
   2.2 Perform initialization of all modules (using command line parameters) with the *init*() function. The *init*() function is administrator specific and is implemented by each administrator type.
3 Start threads (*start*() function).
4 Wait for threads to stop.
5 Perform cleanup.

5.5.1.3 Administrator CORBA interface

The *Administrator* implements three functions from its CORBA interface. They are: *waitToStop*(), *removeNode*() and *hardClose*(). This functions are ways of stopping an administrator. They are abstract functions and should be implemented for each concrete administrator class.

The *waitToStop*() function implementations should just wait for all threads to stop, doing nothing to speed this. *removeNode*() implementations perform a soft close of the node. The node is cut off from its input source, processes all the data it has buffered, and when all buffers are empty, it stops and no data is lost (Req. 25). *hardClose*() is a hard close implementation (Req. 24), this should stop the node as soon as possible. The process has to unregister cleanly from the framework but all data that is buffered is lost.

These three functions operate mainly on thread objects: *Reader* and *fpEB*. They have their own implementation of these three functions. *EventDecomposer* is an exception as it also has to implement them. It is run by the same thread that manages the analysis so its nearly as if it was a thread object.

5.5.1.4 Accessing the administrator

As the administrator has control over the whole node and also has access to the Controller, we decided that it should be available from any place in the process by making it Singleton-like. Singleton is a design pattern which in general means that there can be only one instance of an object and that it can be accessed as if it was a global, static object (but it isn't). There are two main uses of this design in the system.

The first use is for critical situations. For example when a reader gets an exception when trying to access a remote repository it can call one of the *cannotReachOutput*(), *cannotReachInput*() functions and the administrator will undertake proper action (for example contact the Controller and get new, valid references to the output or input repository).

The second use has to do with the problem of fitting the *frspec*, *fwspec* and *fcspec* functions into an object oriented design. We decided that the easiest way to connect these functions with the rest of the system would be through the administrator. The *put*() function is called from *fwspec* to pass data to the *SynchronizedPipe*, and the *get*() function from *frspec* to read data from the event decomposer. This allows us to have only one implementation of the *frspec* and *fwspec* functions. They just call *put*() and *get*() and where the data goes or comes from depends on the administrator they are currently working with.

5.5.2 The *EntryAdmin* class

The input nodes consists of three main components: the event builder, the repository, and the module for reading data. The name of the administrator of input nodes is *EntryAdmin*. Its responsibilities are: initialization of the repository and initialization of the event builder. Two classes derive from *EntryAdmin*: *InputAdmin* and *OrbiAdmin*, see Fig. 33. They have been created to satisfy Req. 8.

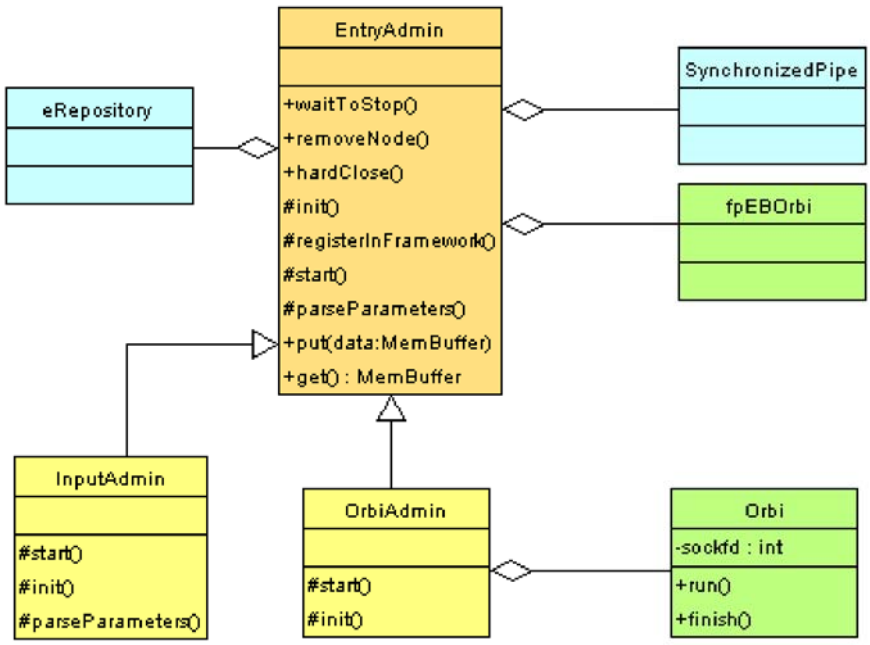


Fig. 33 . EntryAdmin class diagram

In *InputAdmin* the reading module is a Fortran program. It reads the name of an FPACK card file as an input parameter or exits if no card file name is given (Reqs. 9 and 10).

In *OrbiAdmin* the data is read through a socket. *Orbi* is a thread object implemented to read data from a socket. Here the requests 11,12 are not fulfilled as the socket address is hardwired in the code instead of being read as an input parameter. The *Orbi* class was not mentioned when main system modules were described, as it is very simple. It implements a *run*() function which reads data from the socket until EOF is reached, and a *finish*() method which terminates the tread before EOF is reached.

Parallelism at *EntryAdmin*. At least two threads are working all the time at *EntryAdmin*. One is reading data (this is either *Orbi* or a Fortran routine for reading data) and passing it to the *SynchronizedPipe*, the other is *fpEBOrbi* reading physical records from the pipe and making events out of them. A thread is also started for each CORBA call made to the repository.

5.5.3 Output nodes

EndAdmin is designed analogically to *EntryAdmin*, it consists of: a repository, an event decomposer and a writing module. *OutputAdmin* and *LoggingAdmin* inherit from *EndAdmin* (see Fig. 34).

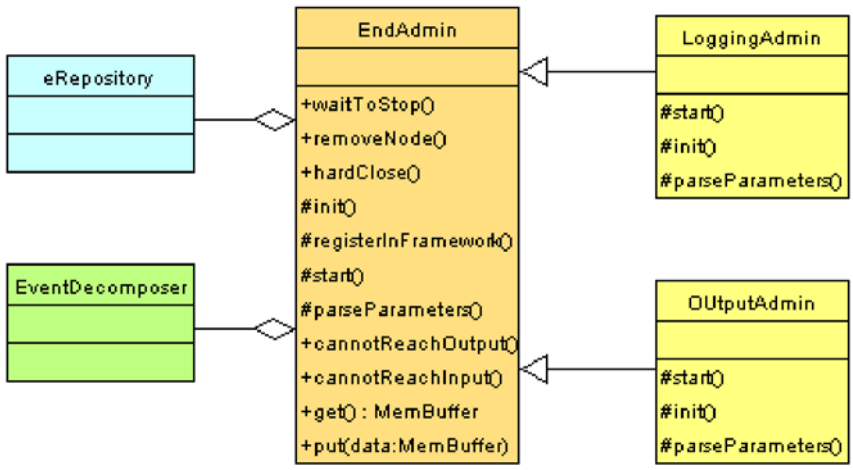


Fig. 34 . EndAdmin class diagram

OutputAdmin has a very simple writing module which just writes the data using a Fortran routine. In *LoggingAdmin* a much more complex procedure is used which extracts additional information from the data before it is written down. This data can be passed over to a file the *BarrierServer* has access to. In such way a callback mechanism is implemented allowing the results to have influence on how the data will be processed. *LogginAdmin* uses external routines and functions to analyze data, which were not part of the project.

Parallelism in *EndAdmin*. One thread is used for reading and processing the result in *EndAdmin*. More threads are created when calls to the repository are made.

5.5.4 Worker nodes

5.5.4.1 Split tasks

The functionality of the worker nodes has been split into two parts. The *JobAdmin* is responsible for the local input and output repositories of the node. They are used to buffer data read from the input node and the results before they are sent to the output node. The initialization process is complex as it includes readers and writers registration (see Fig. 35).

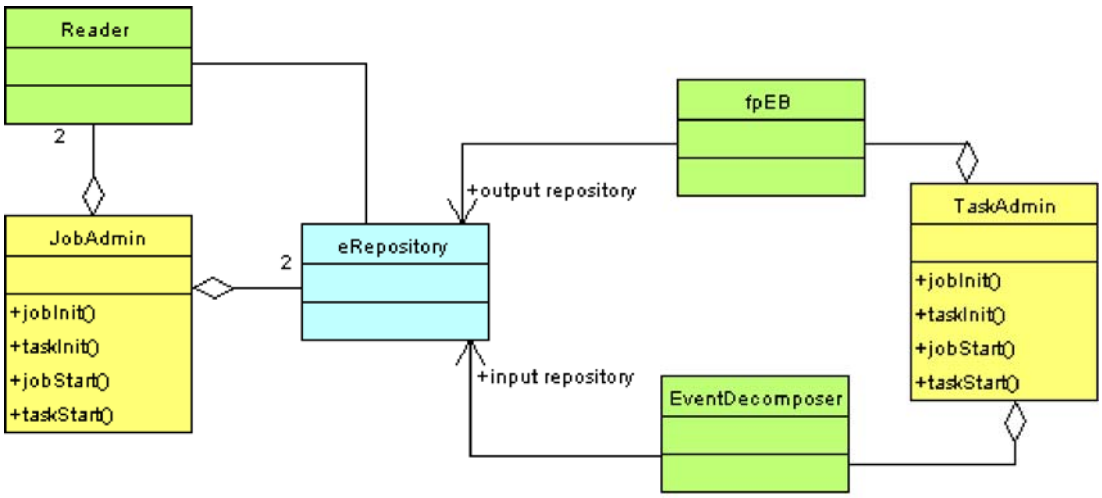


Fig. 35 . JobAdmin and TaskAdmin class diagram

TaskAdmin is complementary to *JobAdmin*. It contains the modules for event decomposition, processing and building. Events are read from the input repository of the *JobAdmin* and fed into the analysis/reconstruction process using the *EventDecomposer*. Results are stored in the Synchronized pipe and read by the event builder like in an input node. The event builder stores them in the output repository of the *JobAdmin*. The system is designed to allow only one event to be inside *TaskAdmin* at a time. A copy of this event is kept in *JobAdmin*. Should the processing program contain a bug and crash or deadlock trying to process it, the event won't be lost but can be written down by *JobAdmin* for further investigation (Req. 21).

The *init* and *start* functions. *JobAdmin* and *TaskAdmin* implement all the abstract functions of the *Administrator* class an administrator should implement. But the *start*() and *init*() functions have been divided into two parts - the 'task' part and the 'job' part. The in 'task' part objects associated with *TaskAdmin* are initialized. For *TaskAdmin*::*taskInit*() just means initialization of local objects and *TaskAdmin*::*jobInit*() initialization of connections to *JobAdmin*. In *JobAdmin*::*jobInit*() initialization of local objects is performed, in *JobAdmin*::*taskInit*() the connections to *TaskAdmin* are initialized, and so on. This is used in *AnalAdmin*.

***AnalAdmin* - reliability vs. performance.** The model described above which uses *JobAdmin* and *TaskAdmin* is very secure and should be always applied when the processing code is not quite well tested. However this solution is slow as it involves additional interprocess communication. For analysis/reconstruction code which is well tested such overhead is unnecessary. For that reason *AnalAdmin* has been designed. It is a combination of *JobAdmin* and *TaskAdmin* which removes the interprocess communication part from them. It is not so secure but much faster.

The *AnalAdmin* has been designed as a template which takes two arguments, *JobAdmin* and

TaskAdmin. This has been done to allow for easy creation of *AnalAdmins* with multiple output support using *MJobAdmin*, *MTaskAdmin* and *MMJobAdmin* classes which are described in next chapter.

Smooth joining of *TaskAdmin* and *JobAdmin* is possible because of dividing the *init*() and *start*() functions into two parts. *AnalAdmin*::*initJob*() just calls *JobAdmin*::*jobInit*() and *AnalAdmin*::*taskInit*() calls *TaskAdmin*::*taskInit*(). *JobAmdin*::taskInit() and *TaskAdmin*::*jobInit*() are obsolete here. The same counts for *taskStart*() and *jobStart*().

5.5.4.2 Multiple output support

For higher input rates or a more complicated logging program one output node may be not sufficient to handle the data flow. For that reason multiple output support has been added. It has been designed in two forms. The first one includes only a slight modification to *JobAdmin*. Instead of the *Reader* class, *MReader* is used. The *MReader* is able to distribute sequences of events to more than one repository. The administrator is called *MJobAdmin* and can be used together with *TaskAdmin*. This classes combined by the *AnalAdmin* template create the *MAnalAdmin*.

The solution mentioned above does not allow for distributing events to different output repositories based on event characteristics. To allow this the events have to be divided at event builder level, before they are stored in the repository. The *MfpEB* class is capable of doing this. *TaskAdmin* which uses *MfpEB* is used *MTaskAdmin*. *MTaskAdmin* can not work together with neither *JobAdmin* nor *MJobAdmin* as they have only one local output repository. *MMJobAdmin* is a *JobAdmin* implementation with multiple output repositories.

Again *MMJobAdmin* and *MTaskAdmin* can be joined using the *AnalAdmin* template. This combination is called *MMAnalAdmin*.

5.6 Additional system modules

In this chapter two modules which were not implemented by me but are vital for the proper functioning of system will be described.

5.6.1 Master and Slave Controller

Master Controller has been designed to have control over the whole system. It can be used to

start tasks on farm machines, to add new machines when the system is running, to cleanup and restart crashed or deadlocked machines. It communicates with nodes via Slave Controllers. There is one Slave Controller per machine in the system. Each Slave Controller communicates with all processes on it's machine and with the Master Controller. They communicate with the administrators through a CORBA interface. The *Controller* design satisfies Reqs. 1,2,22,23.

The *Controller* was being implemented by Alan Campbell, Marcin Kuta [22] and Max Vorobiev [23] who finished the implementation. I have helped with the design.

5.6.2 Logger

The Logger is a logging module for the system. As most of the processes run detached from a terminal their messages would be lost. There is one Logger for the system and it communicates with all nodes via CORBA. Each administrator is equipped with a RedirectionManager and a SafeLog. The RedirectionManager when initialized redirects standard output and error so that all messages are caught and send to the Logger. SafeLog can be used to explicitly log messages in the Logger, whereby it is possible to adjust priorities to the messages. The Logger has been designed and implemented by Grzegorz Mazur [24].

6 Implementation

6.1 Programming language

C++ has been chosen as the basic programming language for implementing the system described in this thesis. Following characteristics have led to this choice:

- Easy access to C and Fortran routines,
- High level constructs like objects, polymorphism, multiple inheritance, templates which allow for a high code reusability and good object oriented design,
- Many ways of optimization,
- Many advanced free programming tools available under linux – gcc, gdb, editor support,
- Many advanced programming libraries.

Accessing Fortran routines. Accessing Fortran routines can be done in C++ by adding an underscore at the end of the routine name and by putting the definition into an extern C clause. An example of this is in the spec.cpp file where implementations of the *frspec*, *fwspec* and *fcspec* functions are placed. Below is an example showing the declaration of the *fwspec_* function.

```
extern "C" {
  void fwspec_(int *LUN, int *NWORDS, int *BUFFER, int *IEND) {
     ...
  }
}
```

Template classes. When the implementation of the *eRepository* class started it seemed that the best approach is to make it as general as possible. The usual approach in an object oriented programming language would be to make a base class *Event* with a basic interface the repository uses and then make all objects or structures which need to be stored in the repository, inherit from this base class *Event*. However this proved to be impossible because events are CORBA structures and can not be made to inherit from some C++ object. The solution to this problem was making the repository a template class where the event and event sequence is a template argument. The definitions of the repository class and associated classes are listed below:

```
template < class ELEM, class SEQ > class UniRepository;
template < class ELEM, class SEQ > class BarrierList;
```

6.2 CORBA implementation

Several open source CORBA implementations for Linux have been considered: ORBacus, omniORB, ACE orb. OmniORB has been chosen for following reasons:

- Small footprint and good performance, better then ORBacus,
- A compact library offering only CORBA as opposite to ACE which is a very complex system of libraries,
- Quick replays from the user list concerning bugs and problems,

We are currently using OmniORB 4.0. See OmniORB web page for further reference [25].

6.3 Threading library

The system being heavily multithreaded choosing a suitable thread library was an important issue. Actually three libraries were considered: omniThreads, JTCThreads and pure POSIX Threads. Pthreads don't suite an object oriented design very well. OmniThreads offer only a simple wrapping about them. Most features are offered by the JTCThread library. Unfortunately using synchronization objects like mutexes and conditionals that are provided by it, has shown to hit performance. As a compromise we decided to use the thread object provided by JTCThread with pure pthread primitives for synchronization which gave us:

- A full featured thread object ,
- Automatic memory management for thread objects,
- Fast synchronization methods.

For more information about JTCThreads see the ORBacus web page [26]. [20]

6.3.1 Thread objects

An object can become a JTCThread by inheriting from the *JTCThread* class as shown in the example:

```
class Reader : public JTCThread {
  ...
}
```

There are three main thread classes in the system: *Reader*, *fpEB*, *Orbi*. *MReader*, *fpEBOrbi*, *MfpEB* are also thread classes because they derive from a thread class.

6.3.2 The *MutexGuard* class

As mentioned before we have decided to use pthread synchronization objects in the system for performance reasons. Two structures were used: *pthread_cond_t* and *pthread_mutex_t*. This bare C structures may cause problems when used in C++ code, especially the mutex. Lets consider a synchronized function:

```
pthread_mutex_t lock;
void foo(){
  pthread_mutex_lock( _lock );

    bar();

  pthread_mutex_unlock( _lock );
}
```

In case the *bar*() function throws an exception the mutex *_lock* will remain locked. One solution is to put a catch statement and unlock the mutex there, but this is unnecessary duplicating of code. The *MutexGuard* class has much simplified the implementation of synchronized functions:

```
class MutexGuard{
public:
  MutexGuard( pthread_mutex_t * lock ):
    _lock( lock )
  {
    pthread_mutex_lock( _lock );
  }
  ~MutexGuard(){
    pthread_mutex_unlock( _lock );
  }
private:
  pthread_mutex_t * _lock;
};
```

It acquires a mutex when constructed and releases it upon destruction. Now the *foo*() function can be written like this:

```
pthread_mutex_t lock;
void foo(){
  MutexGuard guard( _lock );
  bar();
}
```

The mutex *_lock* will be released no matter if an exception is thrown or we just exit the function.

6.4 Singleton design for the *Administrator* class

The *Administrator* implementation should combine two features. First it should allow for an

singleton-like access to the class, second choosing one administrator class from all classes that have been linked to the executable should be possible. It would be also good to have lazy initialization (that is the singleton instance is created when accessed for the first time and not at program startup) as the administrator may contain CORBA objects which should not be initialized before the *main*() function is called. This has been implemented with the ClassRegister template class. Administrator classes can be added to the class register simply by placing following statement in the source file:

```
CReg< Administrator, InputAdmin > CReg< Administrator, InputAdmin
>::cr( "InputAdministrator" );
```

In this example *InputAdmin* is added to the register, and stored under the name "InputAdministrator". No modifications to the *InputAdmin* code are necessary. *Administrator* is given as the base class of *InputAdmin*.

Now to chose an administrator for the program we have to call:

```
ClassRegister< Administrator >::selectSingleton( adminType);
```

where *adminType* is a string with the administrator name ("InputAdministrator" for example). The instance of the class will be created when the singleton is first accessed:

```
Administrator *a = ClassRegister< Administrator >::getSingleton();
```

Subsequent calls to *getSingleton*() will return the same object. This is used for example in *frspec_* implementation:

```
void fwspec_(int *LUN, int *NWORDS, int *BUFFER, int *IEND) {

  ClassRegister<Administrator>::getSingleton()->
  put(MemBuffer(*NWORDS * WORD_SIZE, BUFFER));

  (*IEND) = 0;
  return;
}
```

6.5 Development tools

The main criteria for selecting development tools was that they are open source or at least free. Developers participating in the project have much freedom in selecting their tools. Here is a list of tools which were arbitrary chosen for the project:

- CVS - 1.10.8 as the version control system [27],
- gcc - 2.95.3 for compiling C++ code [28],
- gdb - 5.0 for debugging C++ code [29],[30]
- Doxygen - 1.2.8.1 to generate class documentation [31],

Additional libraries:

- Stlport 4.5 - a thread safe STL library implementation [32],
- CmdLine - a C++ for parsing command line options and configuration files [33],

Some attempts to use a project manager like KDevelop or Source Navigator were made but we gave it up.

7 Tests and validation

The system in its current form was ready in summer 2002. Since then many performance and feature tests have been performed to check if its ready for data taking from the experiment.

Tested feature: Adding and removing worker nodes at runtime Req. 1,2
Configuration: 1 input node reading from a 1GB data file 1 output writing data to dev/null 6 worker nodes - *AnalAdmin* - configured to stop processing and unregister from the system after 400 events. No real processing was done, only passing data to output.
Realization: Each node was restarted immediately after it stopped.
Results: Passed.

Tested feature: Coping with an input rate of 10 MB/s Req. 5
Configuration: 1 input node reading from a 1GB data file 1 output writing data to dev/null 3 worker nodes - *AnalAdmin* - No real processing was done, only passing data to output.
Realization: Before stopping the input node printed performance information.
Results: Not passed. A processing speed of about 6-7 MB/s was reached
Comments: This test was done in summer 2001. Its result was the reason for adding multiple input support to the system.

Tested feature: Processing online data Req. 11
Configuration: 1 input node reading experiment data from a socket 1 output writing data to dev/null 3 -5 worker nodes - one *JobAdmin* connected with two *TaskAdmins* - running H1 filtering/reconstruction program.
Realization: The system was stopped after some time.
Results: Passed. The data was processed. Events containing errors were found on the input side and thrown away.

Tested feature: Stability test, also automatic crashed nodes restarting, retrieving of events which caused the crash Req. 21.
Configuration: 1 input node reading experiment data from a socket 1 output writing data using the logging system 3 -5 worker nodes - one *JobAdmin* connected with two *TaskAdmins* running H1 filtering/reconstruction program.
Realization: The system was running for two weeks. Crashed nodes were restarted automatically by the Controller, the events which caused the crash, could be recovered and save for further investigation. The system had to stop because of a fatal error in the logging process.
Results: Passed.
Comments: The system proved to be the most stable part of the reconstruction framework.

Tested feature: Multiple input support, testing the *BarrierServer*.
Configuration: 2 input nodes reading from two data files. 1 output writing data to dev/null 1 worker node - *AnalAdmin* - for each input node just passing data to the output
Realization: In the files run start/stop records were missing.
Results: Passed. The barriers were distributed properly, missing run start/stop records were recreated

Tested feature: Throughput requirements Req. 5,6
Configuration: 2 input nodes reading from two data files. 1 output writing data to dev/null 2 worker nodes - *AnalAdmin* - for each input node just passing data to the output
Realization: Data was read from a file.
Results: Passed. A throughput of about 15 MB/s was reached

The tests have shown that the system is fully functional and ready to use. The support for multiple input nodes has been necessary to fulfill the most important requirement of the system - enough throughput to process online data. The barrier algorithm is working allowing for smooth insertion and removal of nodes. The barrier server is handling barrier insertion into the data flow correctly. The system is stable and crashed nodes can be restarted quickly.

In the first stage of the tests bugs were found in the code as well as minor changes in the

design were necessary. But due to the modular architecture of the system all problems could be found and removed quickly. The tests which followed did not reveal any new bugs in the system.

8 Summary and future plans

Feature tests have proven that the system is fully functional and stable. The barrier approach has been a success and we were able to address most of the H1 computing problems the previous framework could not handle.

There are currently no new feature requests or known bugs in the system. Future plans include minor refactoring, cleanup of the code and better documentation. This thesis was a step towards creating a full manual of the system. However the system is to large to be described fully in this thesis and I was forced to give up going into details of the implementation.

Still not enough tests, especially scalability tests, have been performed. The system was tested with up to ten worker nodes and two input nodes. The results of tests for bigger configurations would be interesting and could reveal some shortcomings of the system.

It is being considered to separate out a problem independent library from the system. The abstract administrator classes, the repository, readers and the barrier server are nearly problem independent. By implementing the event builder, event decomposer, the reading, writing and processing units, the system can be turned into a framework for processing any embarrassingly parallel problem, offering an interesting synchronization mechanism.

The concurrent character of the barrier algorithm makes it extremely difficult to prove its correctness. By performing tests we can never be sure that all situations have been covered. Creating a more formal model (for example with Petri Nets [34]) of the algorithm would perhaps help to find a limited set of conditions the system has to fulfill in order to be correct. Then by creating unit tests for this conditions we could test the system after any changes to the code. However we can never be sure that the theoretical model we create corresponds to the implementation.

References

[1] DESY homepage, http://www.desy.de/html/home/index.html

[2] DESY, HERA, "A proposal for a Large Electron-Proton Colliding Beam Facility at DESY" , 81-10 (1981)

[3] H1 home page, http://www-h1.desy.de/

[4] Abt et al., "The H1 detector at HERA ",Nucl. Instrum. Meth.,386(1997),310

[5] E. Elsen, "The H1 trigger and data acquisition system", Internal note, H1-01/93262

[6] V. Blobel, "The BOS system. Dynamic memory management.", DESY, 1977

[7] W.J. Haynes, "VMEXI_SSP:VMEtaxi system software package" ",DESY,Version 4.2(1993),

[8] Information about VMETaxi data acquisition by W.J.Haynes, http://www-h1.desy.de/h1/www/h1det/tracker/sitracker/sidaq/SiVMExi/SiVMExi.doc.html

[9] H1 detector web page, http://www-h1.desy.de/general/home/intra_home.html

[10] H. Krehbiel, "The H1 trigger control system ",,DESY(1989),

[11] LynxWorks homepage, http://www.lynuxworks.com/

[12] FPACK users manual, http://www.desy.de/~blobel/fpack.txt

[13] PBS homepage, http://www.openpbs.org/

[14] A. Campbell et al., "Proposal to Merge Level-4and Level-5 Systems of the H1 Experiment", Internal note, H1-01/97-509

[15] Condor web page, http://www.cs.wisc.edu/condor/

[16] Jakub T. Mościcki, "DIANE - Distributed Analysis Environment", CERN, Project report, August 2002

[17] Alan Campbell, "L4/L5 Status report", DESY, February 2000

[18] CORBA standard home page, http://www.corba.org/

[19] Alan Campbell, campbell@mail.desy.de

[20] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell, "Pthreads Programming ", 1996, ed. O'Reilly

[21] M. Ben-Ari, "Podstawy programowania współbieżnego (Principles of concurrent programming)", 1996, ed. Wydawnictwa Naukowo-Techniczne

[22] Marcin Kuta, kuta@ernie.icslab.agh.edu.pl

[23] Maxim Vorobiev, vorobiev@mail.desy.de

[24] Grzegorz Mazur, mazur@chemia.uj.edu.pl

[25] OmniORB web page, http://omniorb.sourceforge.net/

[26] Orbacus homepage, http://www.iona.com/products/orbacus_home.htm

[27] CVS homepage, http://www.cyclic.com

[28] gcc homepage, http://gcc.gnu.org/

[29] gdb homepage, http://www.gnu.org/software/gdb/

[30] Ian Foster, "Designing and Building Parallel Programs", http://www-unix.mcs.anl.gov/dbpp/

[31] Doxygen homepage, http://www.stack.nl/~dimitri/doxygen/

[32] STL port homepage, http://www.stlport.org/product.html

[33] CmdLine homepage, http://www.cmcrossroads.com/bradapp/ftp/src/libs/C++/CmdLine.html

[34] Peter H. Starke, "Sieci Petri (Petri Nets)", 1987, ed. PWN, Warszawa

[35] Alan Campbell et al., "A Framework for Event Filtering and Reconstruction for the DESY H1 Experiment ",Proceedings of Cracow Grid Workshop,(2002),117

List of Figures

List of Tables

Appendix A

Opinion of the Project leader

From: "Alan Campbell" <campbell@mail.desy.de>
To: "Jacek Nowak" <jnowak@ernie.icslab.agh.edu.pl>
Sent: Tuesday, May 20, 2003 5:23 PM
Subject: Re: L45 an opinion

On the Work of Jacek Nowak

Dear sir or madam,

It is a pleasure for me to express my opinion on the masters thesis work of Jacek Nowak.

Jacek joined the L45 upgrade project at an early stage. The main requirements had been formulated and initial tests had been made to verify that the chosen technology (C++ and CORBA) could deliver sufficient performance. Jacek developed the entire implementation of the core data flow framework including barriers as synchronisation elements.

>From the beginning he has produced excellent C++ code and kept it maintainable and understandable through the use of well thoughtout object design and the application of design patterns. He has mastered the techniques required in multithreaded programs and the appropriate usage of advanced features of C++ such as templates and the standard library. He has acquired detailed knowledge of CORBA and its C++ binding.

The project has been very successful. The code is in production use since early 2002, successfully managing the continuous online data processing and filtering of H1 detector data. The excellent stability of the system proves the correctness and quality of Jacek's code.

In addition to these technical abilities, Jacek's very pleasant manner, excellent discussion proficiency, thoughtful thinking and good presentation ability
enabled him to work efficiently in our international collaboration. This greatly helped in the formulation of detailed requirements and in the discussion of implementation options, and enabled other members of the team to learn a lot from Jacek's knowledge and experience.

Yours sincerely,

Dr Alan Campbell